SANDIA REPORT

SAND2003-4404
Unlimited Release
Printed December 2003

Evaluation of TCP Congestion Control Algorithms

Robert M. Long

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2003-4404
Unlimited Release
Printed December 2003

Evaluation of TCP Congestion Control Algorithms

Robert M. Long
Advanced Networking Integration Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-0806

Abstract

Sandia, Los Alamos, and Lawrence Livermore National Laboratories currently deploy high speed, Wide Area Network links to permit remote access to their Supercomputer systems. The current TCP congestion algorithm does not take full advantage of high delay, large bandwidth environments. This report involves evaluating alternative TCP congestion algorithms and comparing them with the currently used congestion algorithm. The goal was to find if an alternative algorithm could provide higher throughput with minimal impact on existing network traffic. The alternative congestion algorithms used were Scalable TCP and HighSpeed TCP. Network lab experiments were run to record the performance of each algorithm under different network configurations. The network configurations used were back-to-back with no delay, back-to-back with a 30ms delay, and two-to-one with a 30ms delay. The performance of each algorithm was then compared to the existing TCP congestion algorithm to determine if an acceptable alternative had been found. Comparisons were made based on throughput, stability, and fairness.

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockeed Martin Company for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This page intentionally left blank

Contents

Figures

Background

Network congestion occurs when a machine (computer, router, switch, etc) receives data faster than it can process. Congestion leads to dropped packets. For connection oriented protocols, such as TCP, since the destination does not receive the dropped packet, no acknowledgement is sent to the sender. Therefore, the sender retransmits the lost packet, possibly leading to more congestion. Network congestion led to several collapses of the Internet in the late 80's.

When TCP was first standardized in 1980 the standard did not include any congestion control mechanisms [1]. TCP congestion control was created to allow TCP connections to recover from a lost packet more gracefully. A congestion window (Cwnd) was implemented to limit the amount of outstanding data (unacknowledged packets) a connection can have at any given time. The congestion window is resized upon receiving an ACK for new data or a congestion event (receiving duplicate ACKs for the same data or a timeout).

The TCP congestion control algorithm used today is based on the algorithms proposed by Van Jacobson in 1988 [2]. This TCP congestion control algorithm was designed over a decade ago for much lower bandwidth speeds than used today. The network speeds used then were around 32Kbps. Sandia, Los Alamos, and Lawrence Livermore National Laboratories are currently connected using a 2.5Gbps network, over 78000 times the speed used in 1988. With the current high speed, long delay networks, an alternative congestion control could possibly provide better utilization of the channel. The alternative congestion control algorithms being tested are HighSpeed TCP and Scalable TCP.

Introduction

Standard TCP

Standard TCP uses the congestion control algorithms described in RFC2581 [3]. The algorithms used are: slow start, congestion avoidance, fast retransmit, and fast recovery. A TCP connection is always using one of these four algorithms throughout the life of the connection.

Slow Start

After a TCP connection is established, the initial congestion algorithm used is the slow start algorithm. During slow start, the congestion window is increased by one segment for every new ACK received. The connection remains in slow start until one of three events occurs. The first event is when the congestion window reaches the slow start threshold. The connection then uses the congestion avoidance algorithm after the congestion window is greater than, or equal to, the slow start threshold. The slow start threshold, ssthresh, is a variable used to determine when the connection should change from the slow start algorithm to the congestion avoidance algorithm. The initial value for ssthresh is set to an arbitrarily high value [3], this value is not usually reached during the initial slow start after a new connection is established. The second event is receiving duplicate ACKs for the same data. Upon receiving three duplicate ACKs, the connection uses the fast retransmit algorithm. The last event that can occur during slow start is a timeout. If a timeout occurs, the congestion avoidance algorithm is used to adjust the congestion window and slow start threshold.

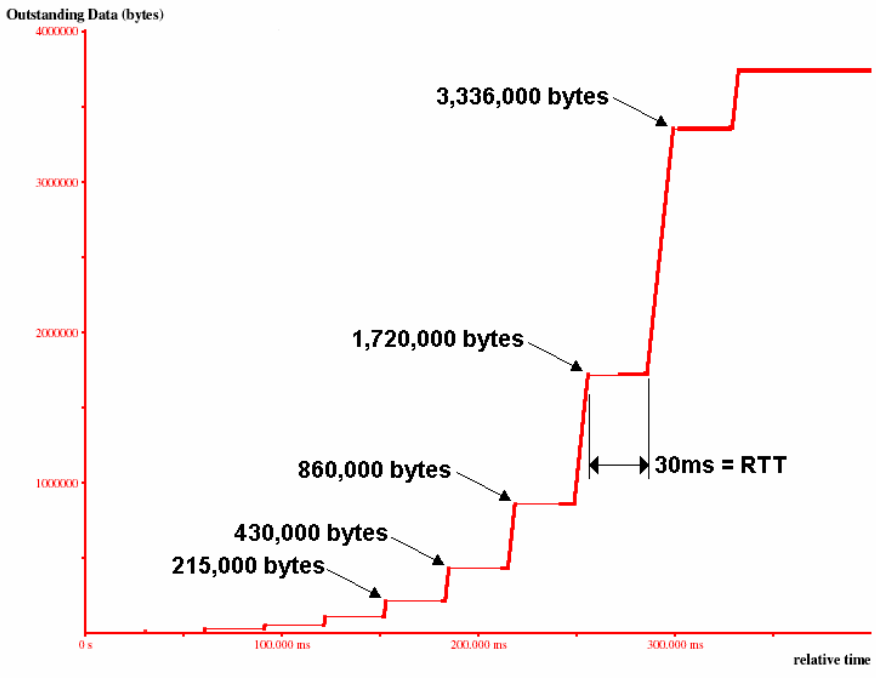

Figure 1. Example of Slow Start.

Figure 1 is an example of slow start for a TCP connection. The 30ms delay between increases is the round-trip time (RTT) for the given example. The figure shows the outstanding data bytes for the connection. Observe how the outstanding data doubles every RTT, this indicates that the congestion window is doubling each RTT. The outstanding data levels off at 3,740,000 bytes. For the given example, this is the maximum outstanding data possible given by the bandwidth-delay product. The bandwidth-delay product is the bandwidth multiplied by the delay; In the case of the example, it is 1Gb/s * 30ms = 30Mb or approximately 3.75MB. The bandwidth-delay product gives you the number of bytes it takes to fill the channel that you are using.

Congestion Avoidance

The congestion avoidance algorithm consists of two parts, additive increase (AI) and multiplicative decrease (MD), referred to as AIMD. When in congestion avoidance, additive increase is used to adjust the congestion window after receiving new ACKs. Multiplicative decrease is used to adjust the congestion window after a congestion event occurs.

Additive Increase

After receiving an ACK for new data, the congestion window is increased by $(MSS)^2/Cwnd$, where MSS is the maximum segment size, this formula is known as additive increase. The goal of additive increase is to open the congestion window by a maximum of one MSS per RTT. Additive increase can be described by using equation (1):

$$\mathbf{Cwnd = Cwnd + a*MSS^2/Cwnd} \qquad \mathbf{(1)}$$

where the value of a is a constant, a = 1.

Multiplicative Decrease

Multiplicative decrease occurs after a congestion event, such as a lost packet or a timeout. After a congestion event occurs, the slow start threshold is set to half the current congestion window. This update to the slow start threshold follows equation (2):

$$\mathbf{ssthresh = (1 - b)*FlightSize} \qquad \mathbf{(2)}$$

FlightSize is equal to the amount of data that has been sent but not yet ACKed and b is a constant, b = 0.5. Next, the congestion window is adjusted accordingly. After a timeout occurs, the congestion window is set to one MSS and the slow start algorithm is reused. The fast retransmit and fast recovery algorithms cover congestion events due to lost packets.

Fast Retransmit

The fast retransmit algorithm was designed to quickly recover from a lost packet before a timeout occurs. When sending data, the fast retransmit algorithm is used after receiving three duplicate ACKs for the same segment. After receiving duplicate ACKs, the sender immediately resends the lost packet, to avoid a timeout, and then uses the fast recovery algorithm.

Fast Recovery

The fast recovery algorithm is used to avoid setting the congestion window equal to one MSS segment after a dropped packet occurs. After a drop occurs, the multiplicative decrease portion of congestion avoidance is then used to update the slow start threshold. Next, the congestion window is set to the new value of ssthresh. After the window size is decreased, additive increase is used to reopen the congestion window.

Figure 2 shows an example of the outstanding data for a TCP connection and illustrates the performance of congestion control. The connection starts using the slow start algorithm until the channel is filled and then switches congestion avoidance. In the example, a single drop occurs around three seconds. The fast retransmit and recovery algorithms are used to resend the lost segment and then cut the congestion window in half. Additive increase is used after the drop to reopen the congestion window.

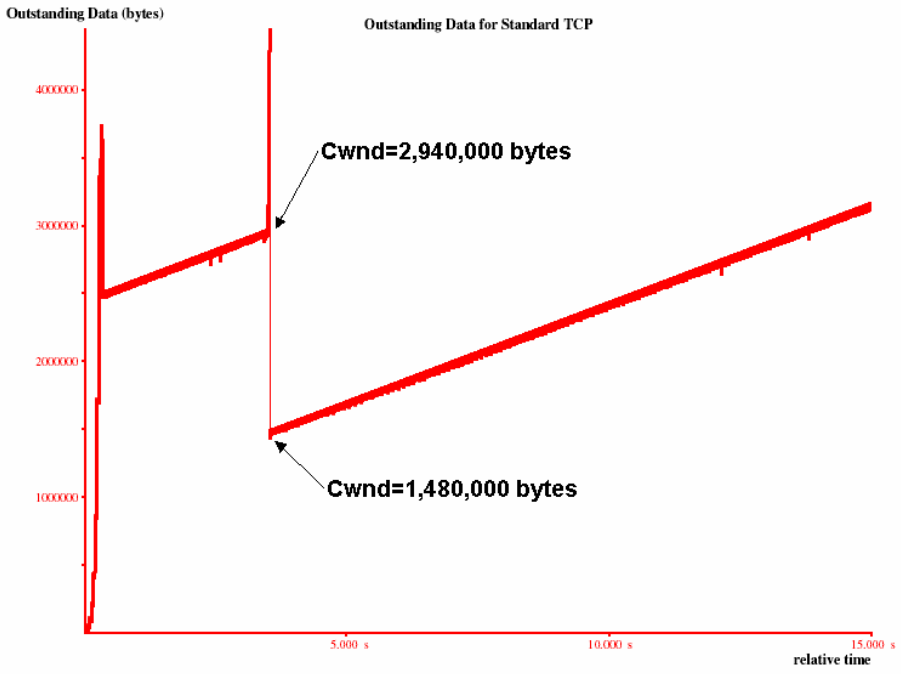

Figure 2. Standard TCP Congestion Control.

HighSpeed TCP

HighSpeed TCP was proposed by Sally Floyd as a sender-side alternative congestion control algorithm [4]. HighSpeed TCP attempts to improve the performance of TCP connections with large congestion windows. Another goal of HighSpeed TCP is to behave similarly to Standard TCP when using small congestion windows.

HighSpeed TCP uses the same slow start and fast retransmit and recovery algorithms that are implemented in Standard TCP. Modifications were only made to the congestion avoidance algorithm.

Congestion Avoidance

HighSpeed TCP still uses an AIMD congestion avoidance algorithm. The changes made involved adjusting the increase and decrease parameters, more specifically the a and b parameters in equations (1) and (2) respectively. The new parameters are found in a table and are based on the current congestion window in MSS segments, given by equation (3).

$$w = Cwnd/MSS \quad (3)$$

The table is created using equations (4) and (5):

$$a(w) = (HW^2 * HP * 2 * b(w)) / (2 - b(w)) \quad (4)$$

$$b(w) = (((HD - 0.5) * (\log(w) - \log(HW))) / (\log(LW) - \log(w))) + 0.5 \quad (5)$$

HW = 83000, HP = 10^{-7}, LW = 38, and HD = 0.1. The first 5 values generated for the HighSpeed TCP table are given below in Table 1.

w	a(w)	b(w)
38	1	0.50
118	2	0.44
221	3	0.41
347	4	0.38
495	5	0.37

Table 1. HighSpeed TCP Table.

Additive Increase

Equation (1) can be rewritten as equation (6) to show HighSpeed TCP's additive increase formula.

$$Cwnd = Cwnd + a(w)*MSS^2/Cwnd \quad (6)$$

The goal of HighSpeed TCP's additive increase is to open the congestion window by a(w) each RTT. Equation (6) allows for large congestion windows to open faster than equation (1) would have allowed.

Multiplicative Decrease

Highspeed TCP follows the same multiplicative decrease algorithm as Standard TCP except for the following change to equation (2). Equation (7):

$$ssthresh = (1 - b(w))*FlightSize \quad (7)$$

The congestion window is resized the same as in Standard TCP after detecting a lost packet, using fast recovery, or a timeout occurs.

Figure 3 shows an outstanding data plot for a connection using HighSpeed TCP and shows how the congestion control algorithms function. The example shows

that the same slow start algorithm is used. A single drop occurs at around 1 second. This lead to the fast retransmit and recovery algorithms being used. At the time of the drop Cwnd ~ 3.6MB. Using equation (3) and Table 1, w = 408 segments so b(w) = 0.37. After the congestion window is reduced to ~2.4MB or w = 262, a(w) = 4. A slight bend in the linear increase is observed around 2.5 seconds. This bend is when w becomes greater than 347 and uses a(w) = 5.

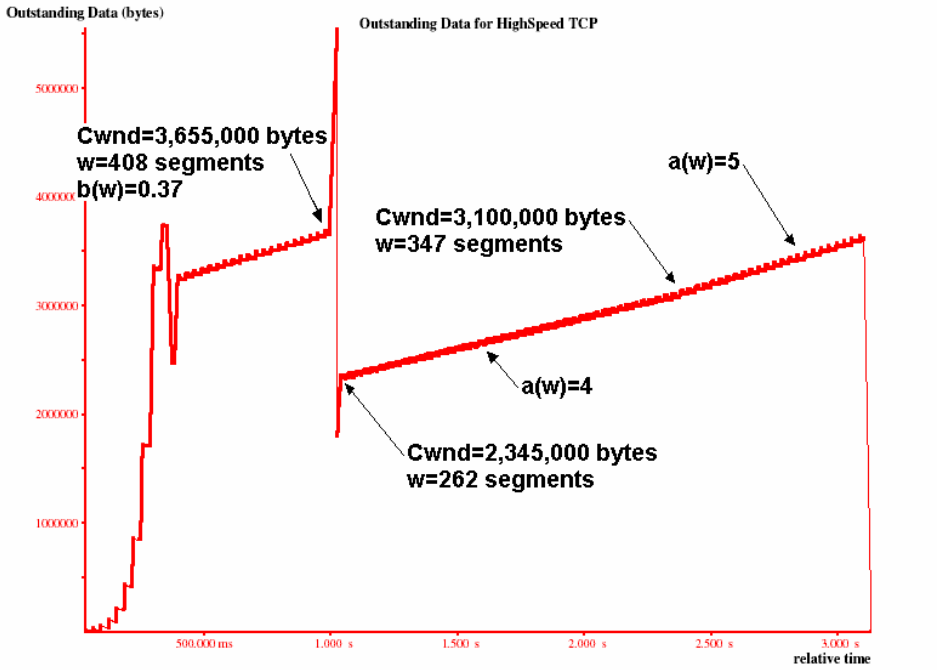

Figure 3. Example of HighSpeed TCP Congestion Control.

Scalable TCP

Tom Kelly proposed Scalable TCP as another alternative sender-side congestion control algorithm [3]. The goal of Scalable TCP is to quickly recover from short congestion periods.

Congestion Avoidance

Scalable TCP uses a different congestion avoidance algorithm than Standard TCP. Scalable TCP uses a multiplicative increase multiplicative decrease (MIMD) rather than the AIMD of Standard TCP.

Multiplicative Increase

The multiplicative increase occurs when the standard additive increase would normally occur. Equation (8) shows the formula used to adjust the congestion window after receiving a new ACK.

$$Cwnd = Cwnd + a*Cwnd \quad (8)$$

where a is adjustable, the value of a used was 0.02.

Multiplicative Decrease
The multiplicative decrease is the same as Standard TCP except that the value of b in equation (2) is adjustable, the value of b used was 0.125.

Figure 4 shows an example of a Scalable TCP connection and the congestion control algorithm that it uses. The connection starts in the slow start algorithm until the channel is filled. Next the connection uses the multiplicative increase portion of congestion avoidance to adjust the congestion window. After a single drop occurs around 1.4 seconds, the fast retransmit and recovery algorithms are used to cut the congestion window by 0.125, the value of b, and congestion avoidance is used again to reopen the congestion window.

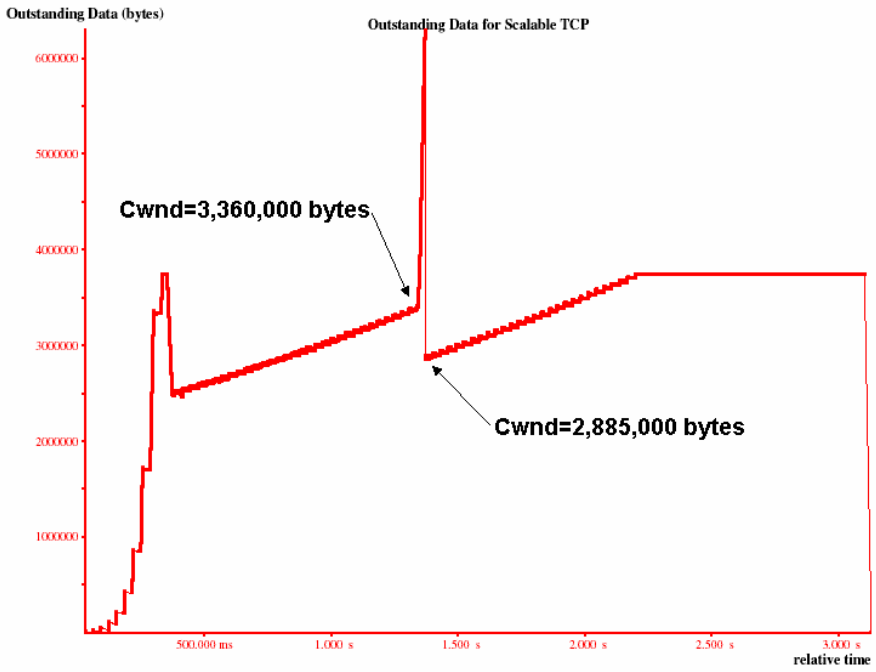

Figure 4. Example of Scalable TCP Congestion Control.

Summary of TCP Congestion Control Algorithms

Table 2 gives an overview of the congestion control algorithms used.

	Increase	Decrease	a,b Values	Algorithm
Standard TCP	a*MSS^2/Cwnd	b*FlightSize	Constant	AIMD
HighSpeed TCP	a(w)*MSS^2/Cwnd	b(w)*FlightSize	Varies	AIMD
Scalable TCP	a*Cwnd	b*FlightSize	Adjustable	MIMD

Table 2. Summary of Congestion Control Algorithms.

Lab Experiments

Test Environment

The test environment consisted of three Dell PowerEdge™ 2650 servers. Each system had dual 2.8GHz Intel® Xeon™ processors, 1GB of memory, and Intel® PRO/1000 Network Interface Cards (NICs). The operating system used in the test environment was RedHat 9. A separate kernel was built for each congestion control algorithm. The kernel used for Standard TCP was a standard 2.4.20. HighSpeed TCP and Scalable TCP each used a modified 2.4.19 kernel. HighSpeed TCP used a 2.4.19-hstcp kernel; the HighSpeed TCP patch was downloaded from [6]. Scalable TCP used a 2.4.19scale0.5.1 kernel; the Scalable TCP patch was downloaded from [7].

The software used to test network performance was Iperf v1.7.0, tcpdump v3.7.2 (with libpcap v0.7.2), tcptrace v6.4.2, and a slightly modified version of Ethereal v0.9.13. Iperf is a networking tool that allows the maximum bandwidth to be measured. Iperf measures the maximum bandwidth by performing memory to memory data transfers. The hardware used was a Spirent Communications Adtech AX/4000™ and a Network Associates Giga-bit Sniffer. The AX/4000™ was used to simulate delays in the network. The Sniffer was used to capture network traffic for analysis.

Capture problems

During the experiments, tcpdump had difficulty accurately capturing network traffic at a full 1Gbps. Tcpdump would drop 5-30% of the packets seen. Since tcpdump was running on one of the computers involved in the test, tcpdump could have also affected the performance of the computer by utilizing system resources. Different capture techniques were used in an attempt to accurately capture network traffic. The goal was to find a capture technique that did not affect the test streams performance or drop packets.

The first alternative capture technique used was the Adtech AX/4000™ capture units. The Adtech capture units satisfied the goal of not dropping packets or affecting the test streams. A conversion routine was written to convert the Adtech capture files to a file format that could be analyzed, such as a tcpdump file. Figure 5 shows a comparison of an Adtech capture vs. a tcpdump capture taken during slow start. The Adtech capture accurately shows more detail than the tcpdump capture. The problem with the Adtech capture units was that they captured the entire packet; capturing the entire packet resulted in only saving half a seconds worth of network traffic. This was determined to be an insufficient amount of time and another technique had to be found.

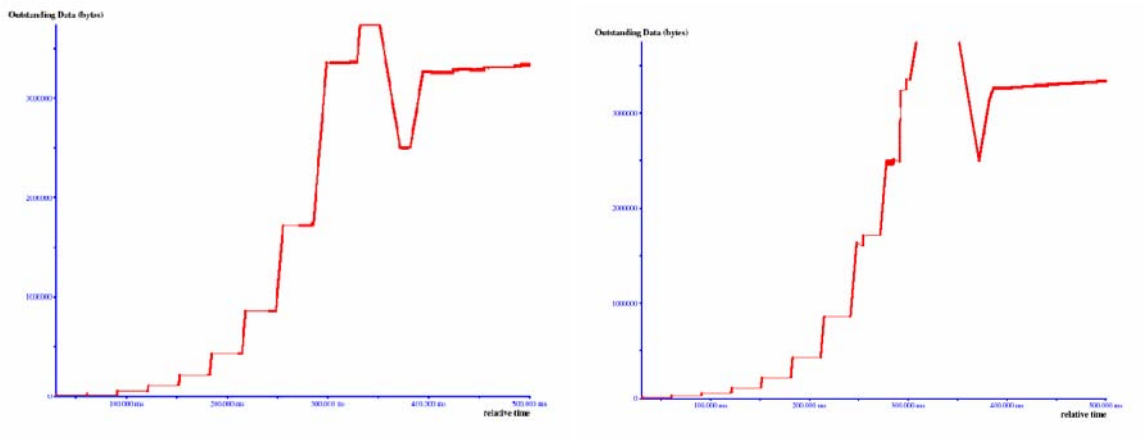

Figure 5. Comparison of Outstanding Data Plots (Adtech-left, tcpdump-right).

The second technique used was a software add-on to the Linux kernel called MAGNET. MAGNET is a kernel event tracker developed at Los Alamos National Laboratories [8]. MAGNET did capture packets successfully. The problem with MAGNET was that a conversion routine was needed to reassemble the events that it captured. MAGNET captured TCP, IP, and socket events separately; to create a tcpdump compatable format these three events would have to be reassembled to create a single event. No statistics were available for packet loss so it could not be determined if the time to write the conversion routine would be worthwhile. In the end, there would have been no way to determine if MAGNET was capturing all the packets and that it was not affecting the processors performance.

The last technique used was a Network Associates Sniffer. The Sniffer satisfied the goals of not dropping packets or affecting the test streams. The capture files from the Sniffer were converted to tcpdump format using Ethereal. It was then noticed that Ethereal did not correctly convert the timestamps. To solve this problem the source code for Ethereal was modified to correctly convert the timestamps. This conversion allowed for Sniffer capture files to be analyzed using tools such as tcptrace. The limitations on the Sniffer were a 72MB buffer, which allowed for capturing around twenty seconds of network traffic.

Tuning parameters

The following tuning parameters were used on both senders and receivers.

```
#configure for jumbo frames
ifconfig eth2 mtu 9000
#using the default value of txqueuelen
ifconfig eth2 txqueuelen 100
#timestamps are on by default
echo 1 > /proc/sys/net/ipv4/tcp_timestamps
#window scaling is on by default
echo 1 > /proc/sys/net/ipv4/tcp_window_scaling
#selective ack is on by default
echo 1 > /proc/sys/net/ipv4/tcp_sack
```

```
echo 8388608 > /proc/sys/net/core/wmem_max
echo 8388608 > /proc/sys/net/core/rmem_max
echo "4096 87380 4194304" > /proc/sys/net/ipv4/tcp_rmem
echo "4096 65536 4194304" > /proc/sys/net/ipv4/tcp_wmem
```

The window sizes were chosen based on the bandwidth-delay product of 3.75MB. The following command was also run before each test was executed.

```
/sbin/sysctl –w net.ipv4.route.flush=1
```

Flushing the route reset the slow start threshold (ssthresh) for all connections. This needed to be done to allow for repeatable test results.

Back-to-Back

The back-to-back test was run to verify that the congestion control algorithms did not affect network performance given ideal network conditions, no delay and no competing traffic. A network diagram is shown in Figure 6.

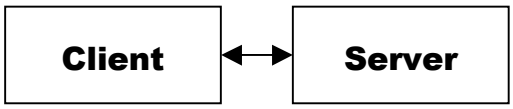

Figure 6. Network Diagram: Back-to-Back.

The results from the back-to-back test showed that the congestion control algorithms behave similarly in this network environment. Figure 7 shows the throughput for the tested algorithms using standard ethernet frames, MTU = 1500, and various window sizes. The results shown are an average of 10 Iperf tests. The Iperf tests used were each run for ten seconds. Scalable TCP shows slightly lower performance for 1MB and 4MB windows. These performance numbers are the result of a single bad run for each of these window sizes. The other nine tests for 1MB and 4MB windows reported throughput numbers of 940Mb/s.

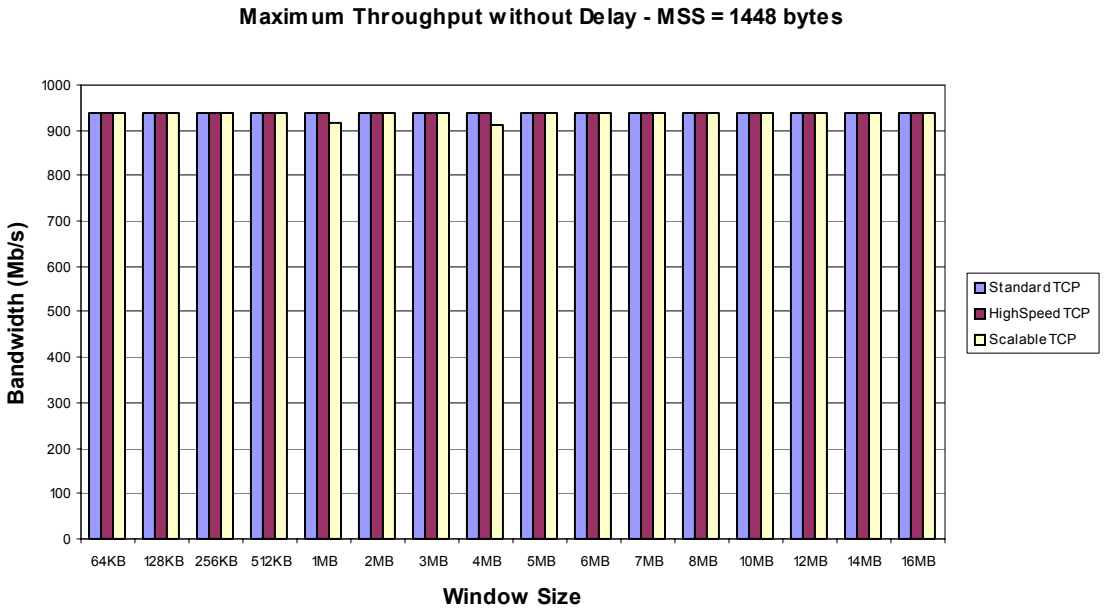

Figure 7. Back-to-Back Throughput - MSS = 1448 bytes.

Figure 8 shows the throughput for the tested algorithms using jumbo frames, MTU = 9000, and various window sizes. For jumbo frames, the three algorithms behaved similarly for all window sizes.

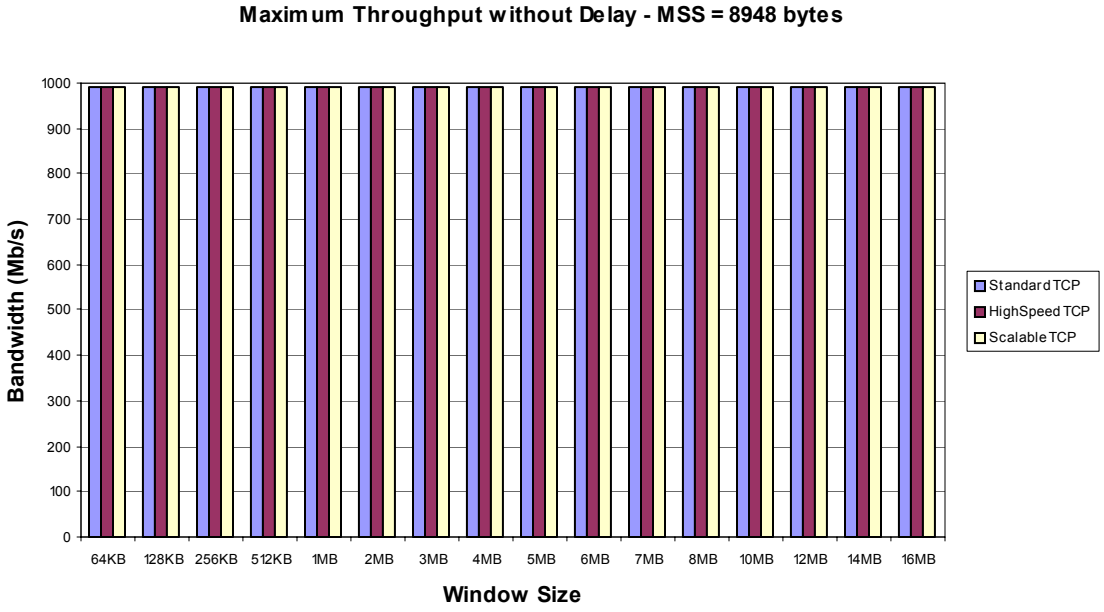

Figure 8. Back-to-Back Throughput - MSS = 8948 bytes.

Back-to-Back with delay

For the back-to-back with delay test a delay of 30ms delay was added to the back-to-back test setup. The 30ms delay was chosen to simulate the RTT from Sandia to Lawrence Livermore. This test allows the behavior of each congestion control algorithm to be observed in the environment, long delay and high bandwidth, that it is being tested for. The back-to-back with delay circuit is shown in Figure 9.

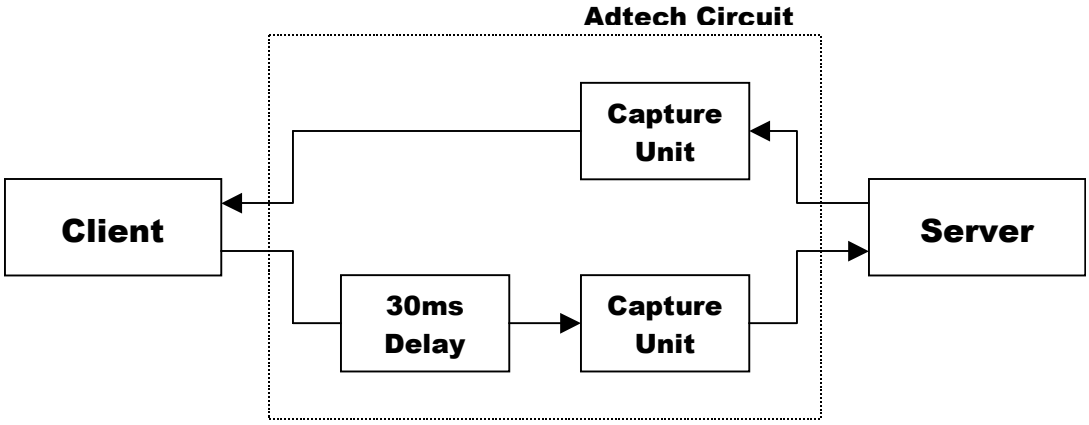

Figure 9. Network Diagram: Back-to-Back with Delay.

Figure 10 shows the results using standard Ethernet frames. The test results shown use the same principles as the back-to-back tests, that is that they are the average of ten-10 second Iperf tests. The results are fairly consistent for each of the tested algorithms. The biggest exception occurs when using 512KB windows. For 512KB windows, HighSpeed TCP performed better than Standard TCP, while Scalable TCP performed worse than Standard TCP.

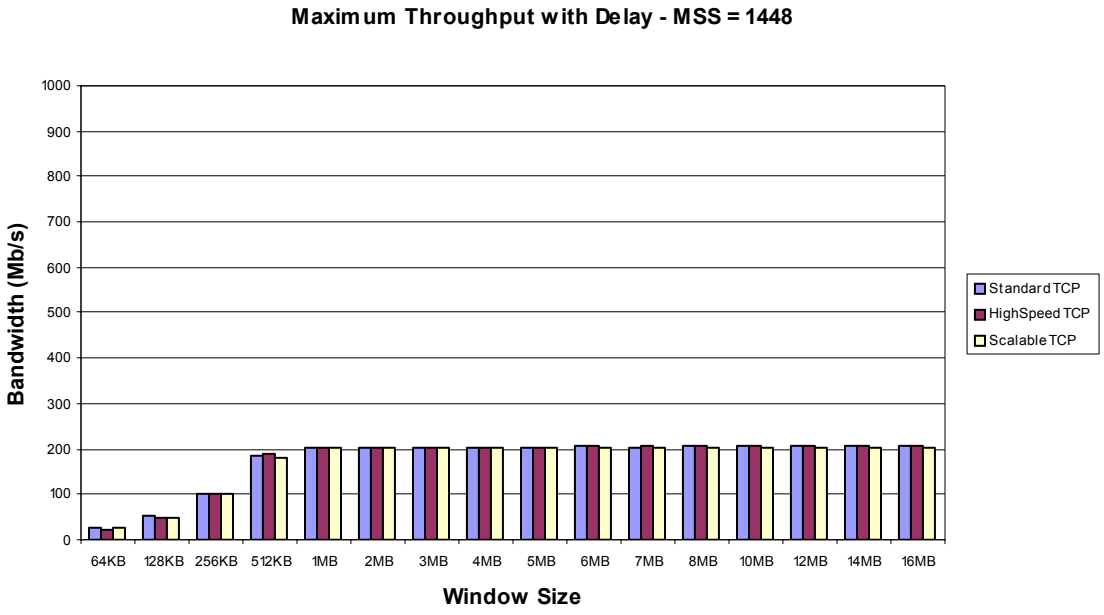

Figure 10. Back-to-Back with Delay Throughput - MSS = 1448 bytes.

Figure 11 shows the results using jumbo frames. In this case, the three algorithms perform the same until 3MB windows. For window sizes greater than 3MB, HighSpeed TCP outperformed Standard TCP by over 100Mb/s, roughly a 14% increase in throughput. Scalable TCP also performed better than Standard TCP by around 70Mb/s, an 8% increase in throughput.

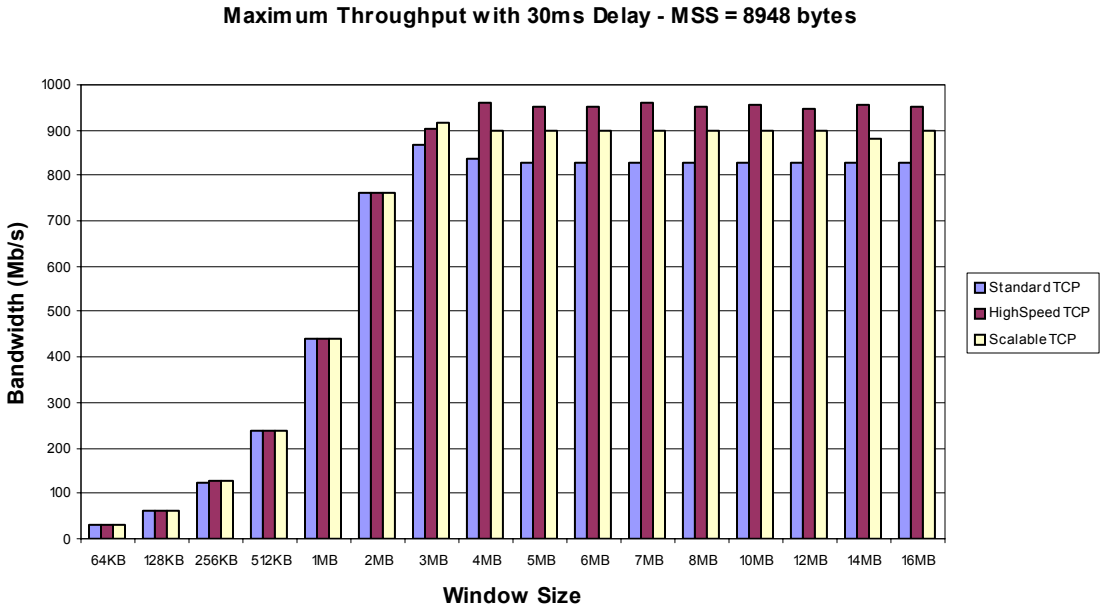

Figure 11. Back-to-Back with Delay Throughput - MSS = 8948 bytes.

The increase in performance is unexpected since the difference between the algorithms is the congestion control mechanisms. Upon further investigation, it was noticed that the test streams are backing off periodically. The back off does not seem to be the result of a network congestion event. The back off for each TCP algorithm is shown in Figure 12.

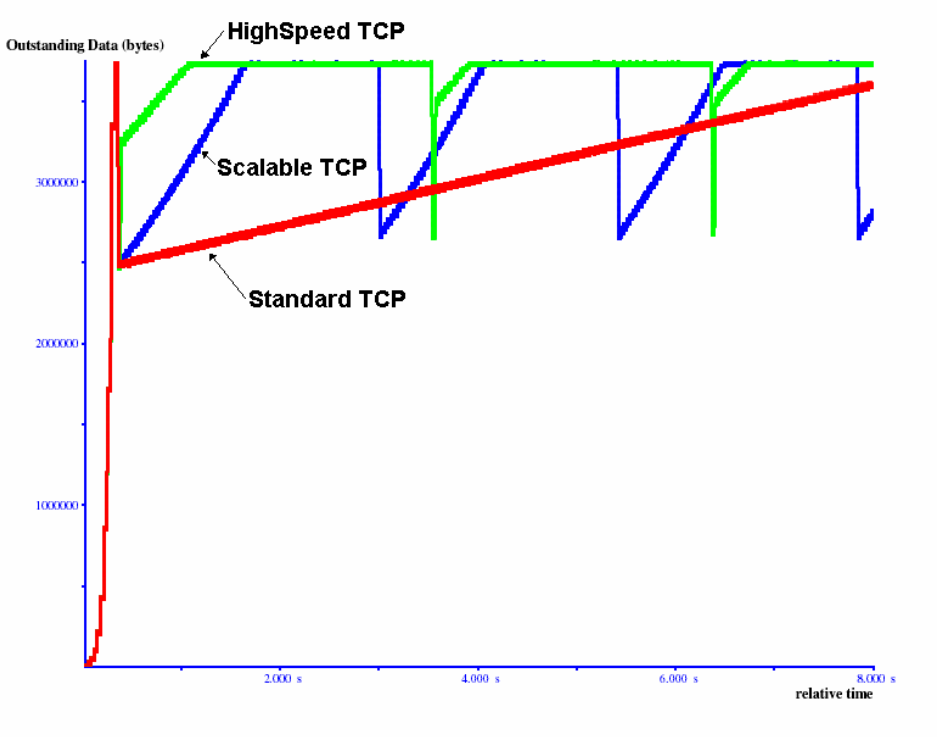

Figure 12. Back off with txqueuelen=100.

Figure 12 shows the outstanding data bytes for a test stream using jumbo frames and 16MB windows. From the figure, each algorithm cuts the window back after slow start ends. It does not appear to be a network congestion event since the outstanding data window after the decrease is equal for all three algorithms. The performance increase for HighSpeed TCP and Scalable TCP occurs because of the way each algorithm recovers from this back off. The back off also seems to be periodic, as shown by HighSpeed TCP and Scalable TCP. The back off also occurs on a regular interval for Standard TCP but this is not shown in Figure 12. By altering the txqueuelen, the back off eventually goes away and all three algorithms obtain the same performance. For the remainder of the evaluation, the default value of txqueuelen=100 was kept and effects of txqueuelen are investigated in a later section.

Two-to-One with delay

The two-to-one test was run to evaluate the fairness of the congestion control algorithms. For the two-to-one test, the 30ms delay was kept for the network setup. The setup used for the two-to-one test is shown in Figure 13.

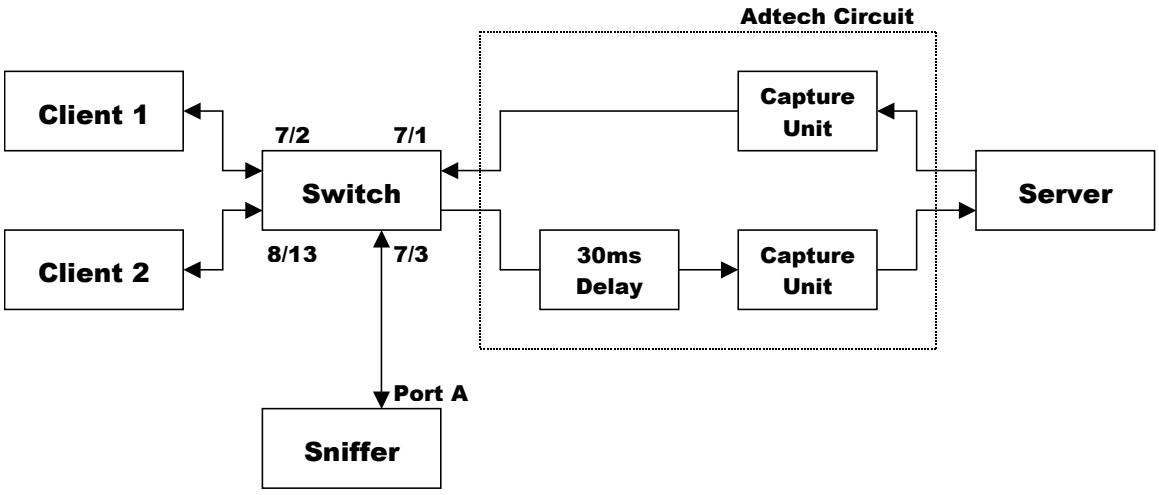

Figure 13. Network Diagram: Two-to-One with Delay.

A Sniffer was used to capture the traffic of the competing streams. The captures were taken to observe how each of the streams share the channel. The following figures show throughput plots for competing streams. Iperf tests were used to generate the test streams; each stream used jumbo frames, 9000 byte MTU's, with 16MB window sizes and was run for 20 seconds. The captures taken are of a single run and show the behavior for different competing streams.

In Figure 14, clients 1 and 2 are using Standard TCP.

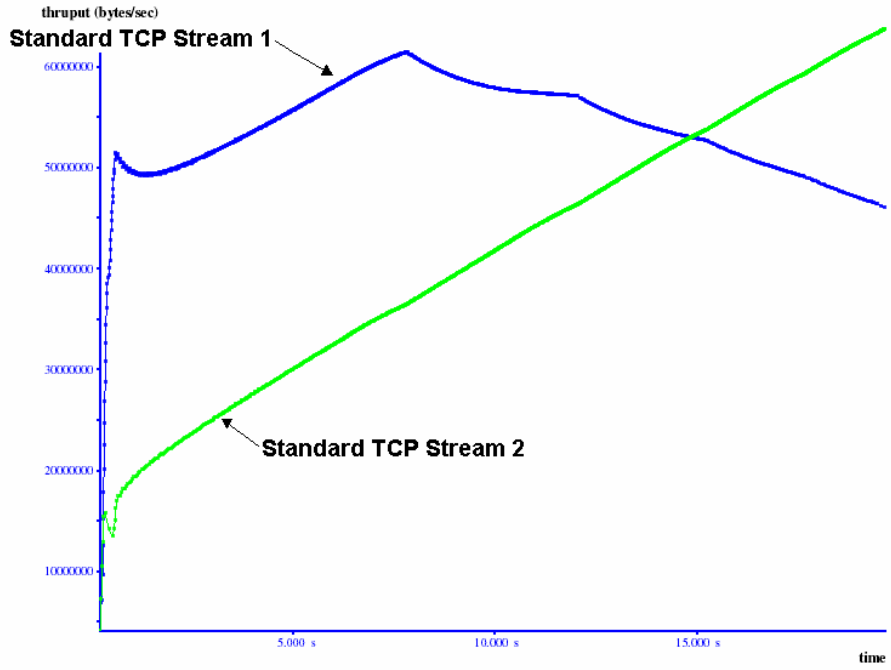

Figure 14. Standard TCP vs. Standard TCP (20 seconds).

To determine if two Standard TCP streams oscillate, tcpdump was used to capture a 240 second test. Tcpdump was used since it was the only capture method that could be used for a 240 second test. The results are shown in Figure 15. From Figure 15, it appears that Standard TCP streams do oscillate, sharing the bandwidth.

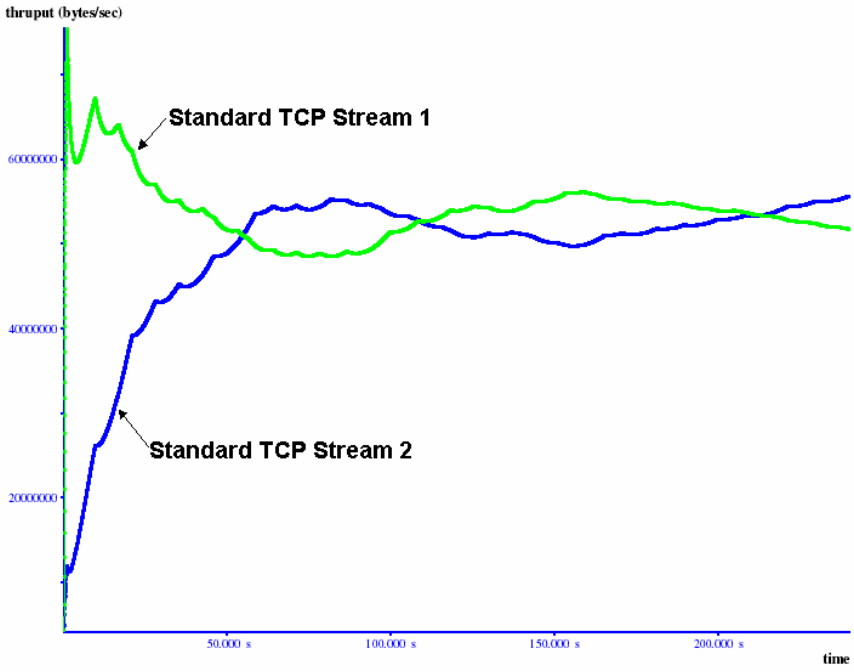

Figure 15. Standard TCP vs. Standard TCP (240 seconds).

In Figure 16, client1 is using HighSpeed TCP and client 2 is using Standard TCP.

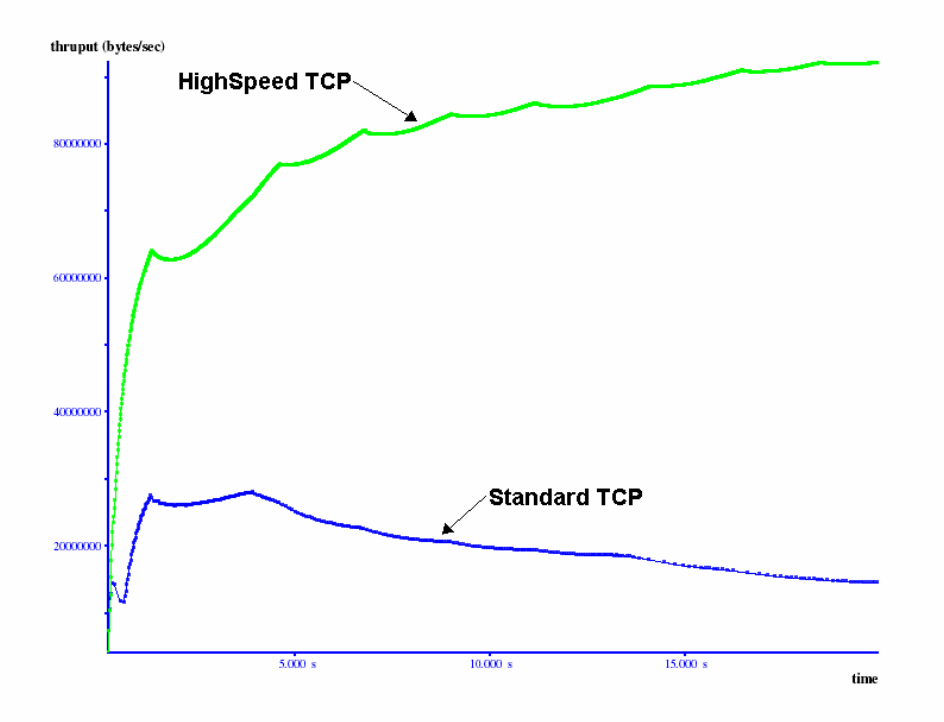

Figure 16. HighSpeed TCP vs. Standard TCP.

In Figure 17, client 1 is using Scalable TCP and client 2 is using Standard TCP. Scalable TCP shares very little of the bandwidth with Standard TCP in Figure 17.

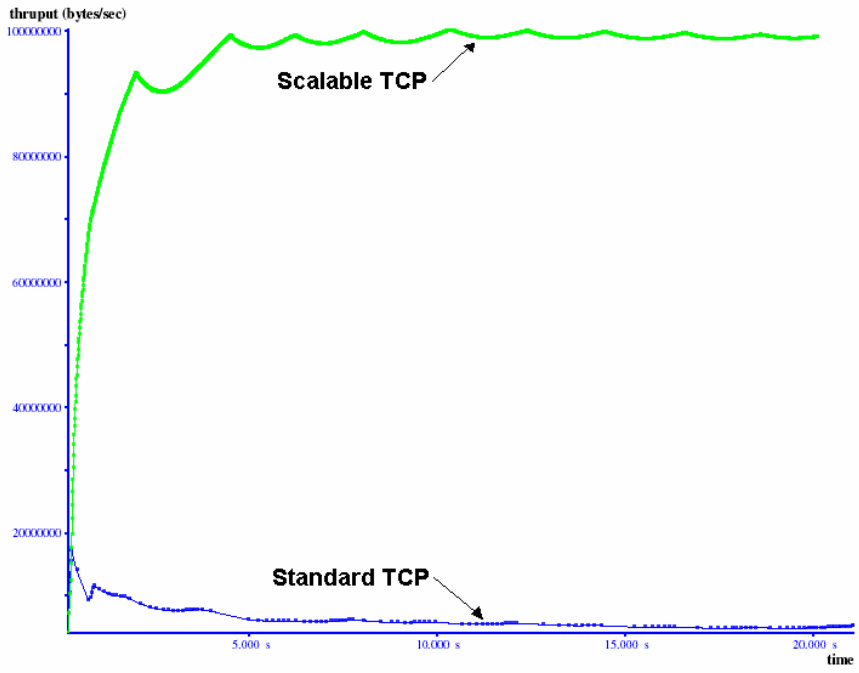

Figure 17. Scalable TCP vs. Standard TCP.

In Figure 18, clients 1 and 2 are using HighSpeed TCP.

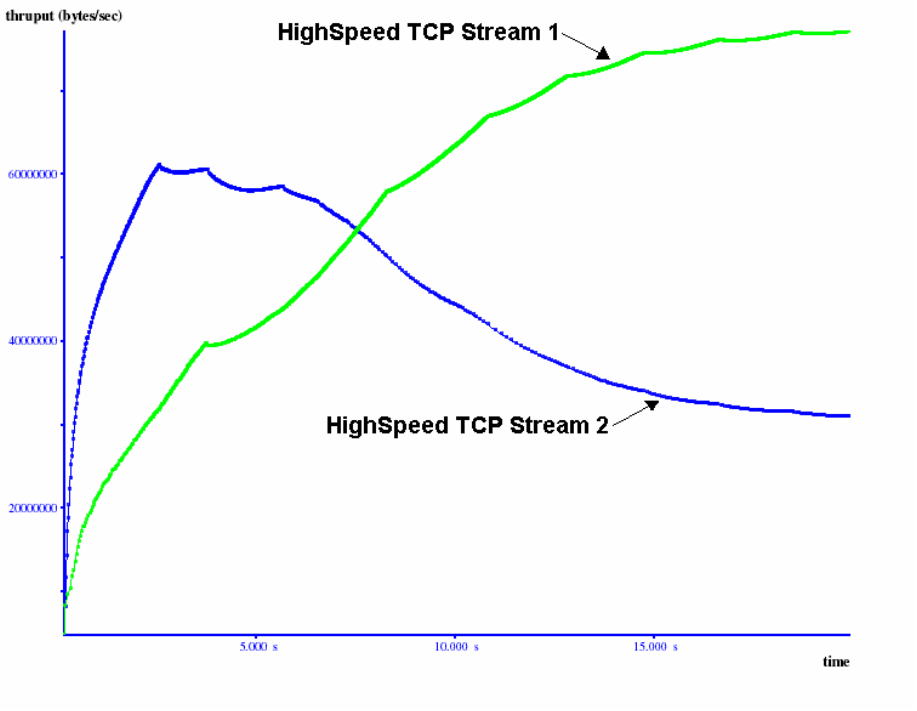

Figure 18. HighSpeed TCP vs. HighSpeed TCP.

In Figure 19, clients 1 and 2 are using Scalable TCP.

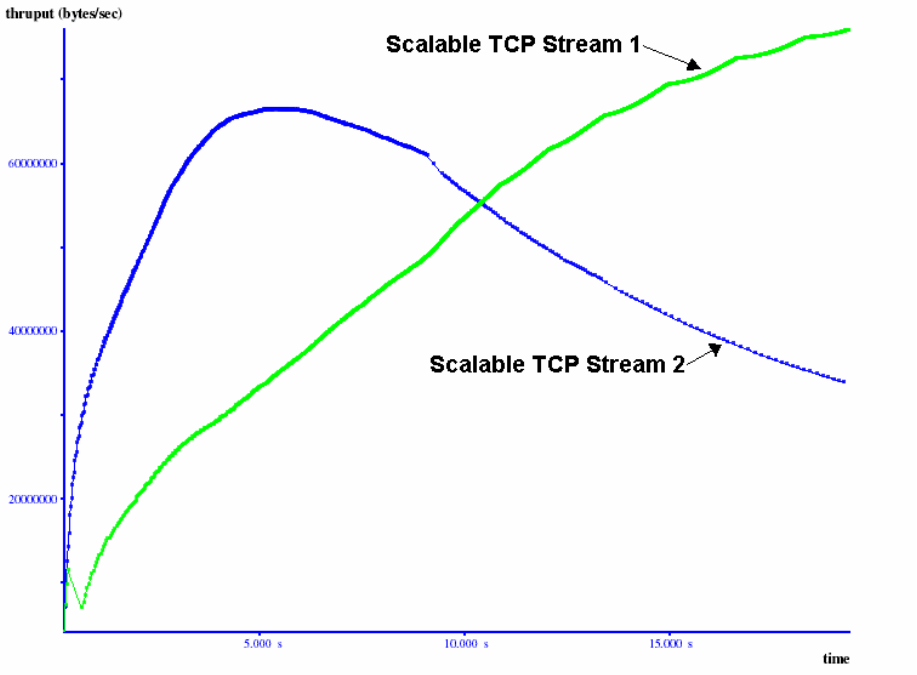

Figure 19. Scalable TCP vs. Scalable TCP.

Longer captures were taken of HighSpeed TCP vs. HighSpeed TCP and Scalable TCP vs. Scalable TCP using tcpdump. In both cases, the captures showed that they oscillate similar to two Standard TCP streams in Figure 15.

Figure 20 shows the average results of ten-10 second Iperf tests using jumbo frames. For window sizes greater than 2MB, the second Standard TCP stream only gets 38% of the channel with an average of 308Mb/s vs. 500Mb/s of the first Standard TCP stream.

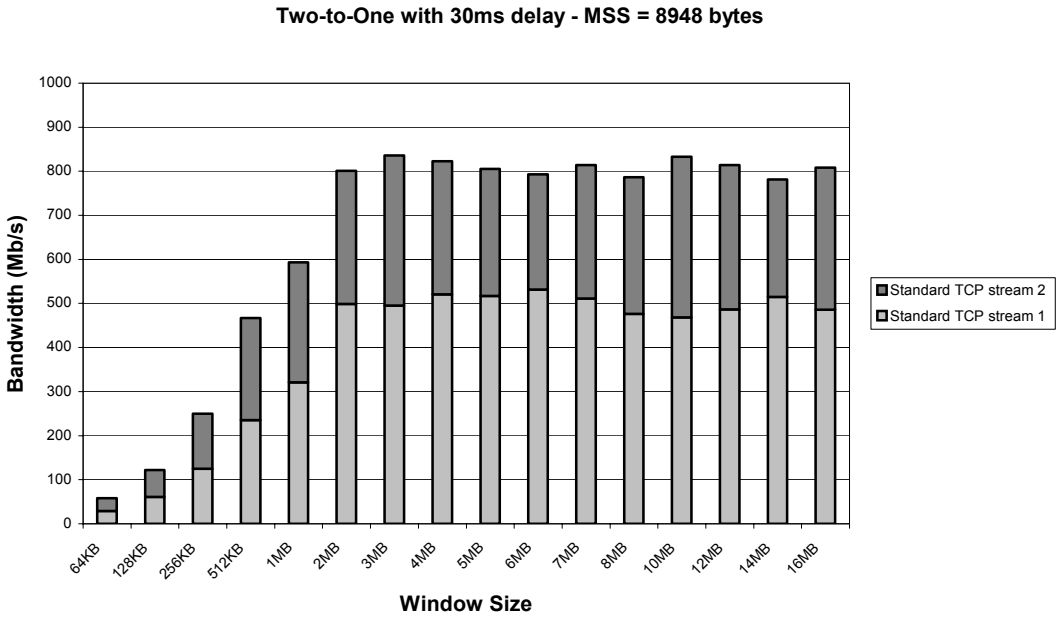

Figure 20. Standard TCP vs. Standard TCP.

Figure 21 shows a HighSpeed TCP stream and a Standard TCP stream using jumbo frames. For window sizes greater than 2MB, HighSpeed TCP allows the Standard TCP stream 35.5% of the channel with an average throughput of 285Mb/s vs. 520Mb/s of HighSpeed TCP.

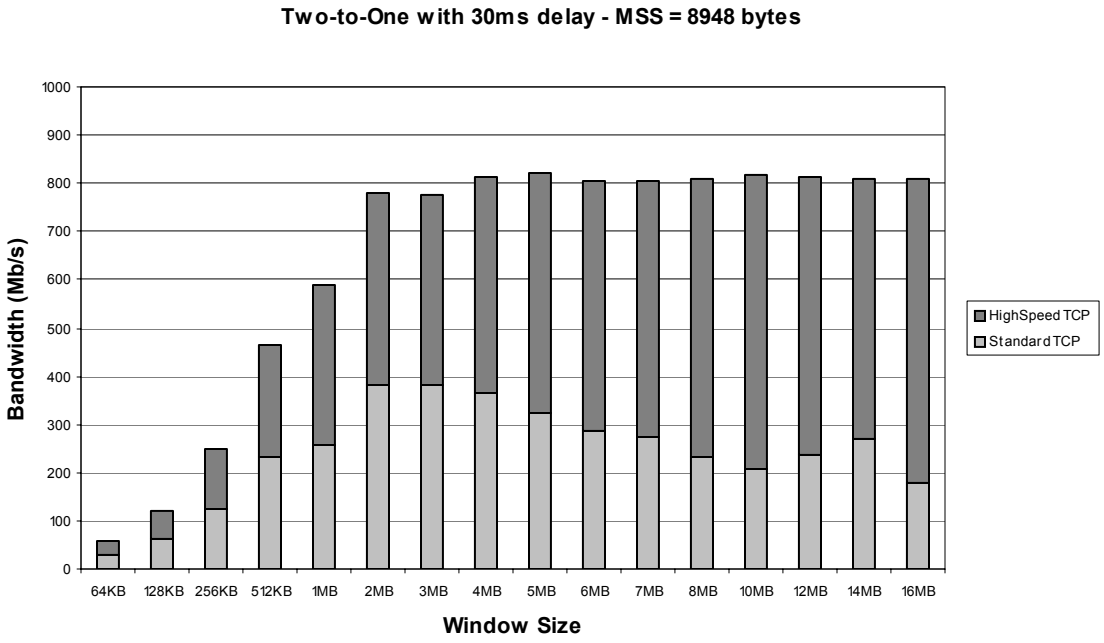

Figure 21. HighSpeed TCP vs. Standard TCP.

Figure 22 shows the throughput from a Scalable TCP stream and a Standard TCP stream. For window sizes greater than 2MB, Standard TCP only gets an average of 29.5% of the channel with 242Mb/s vs. 582Mb/s of Scalable TCP.

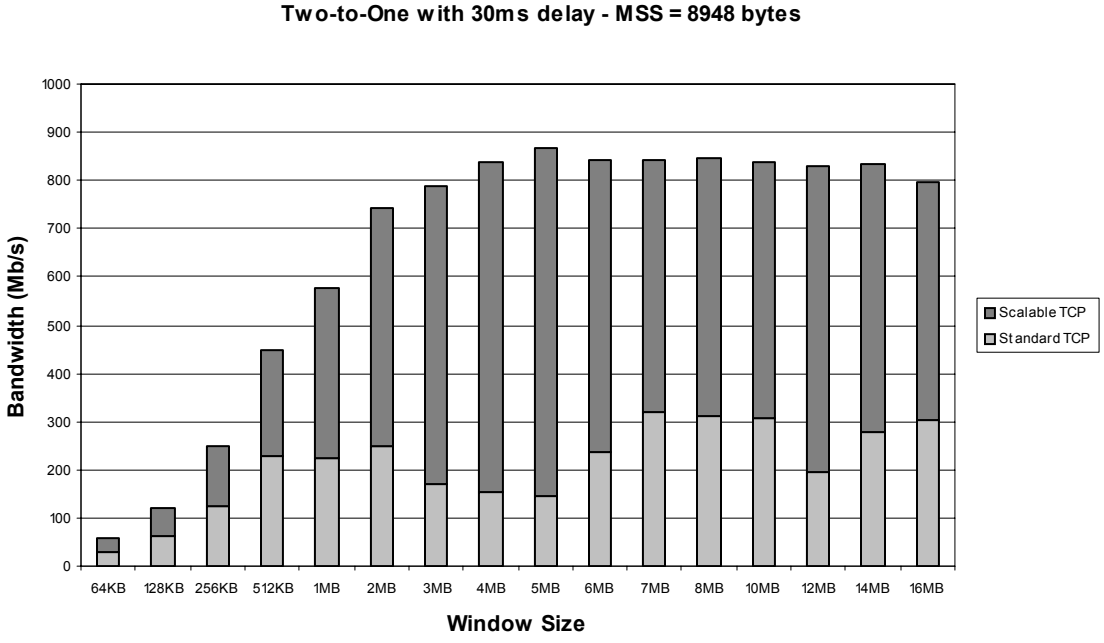

Figure 22. Scalable TCP vs. Standard TCP.

Two-to-One testing shows the fairness between two competing streams. For ten-10 second Iperf tests, Standard TCP shares 38%, HighSpeed TCP shares 35%, and Scalable TCP shares 29.5%.

Adjusting Scalable TCP a,b Values

Scalable TCP uses adjustable values for the increase and decrease parameters, a and b used in equations (2) and (8). The current values used were the default values included in the Scalable TCP Linux kernel, a=1/50 and b=1/8, providing an increase of 2% each RTT and a decrease of 12.5% on a congestion event. Alternative a,b values could continue to provide higher overall throughput and possibly increase the fairness with existing traffic. The alternative a,b values were chosen to keep the a/b ratio currently used, a/b=(1/50)/(1/8) = 4/25, and the a/b ratio described by Tom Kelly in [5], a/b=(1/100)/(1/8) = 2/25. For each a/b ratio, four different a,b sets were chosen. The sets will be described in [a,b] notation. For a/b=2/25, the following values were used [1/25,1/2], [1/50,1/4], [1/100,1/8], and [1/200,1/16]. For a/b=4/25, the following values were used [2/25,1/2], [1/25,1/4], [1/50,1/8], and [1/100,1/16].

Two-to-one with 30ms delay tests were run using each [a,b] pair. The results would allow to observe the effects that each [a,b] pair had on the overall combined throughput and the fairness between streams. Figures 23-30 show the results for various window sizes using MSS = 8948 bytes for each [a,b] pair. The results are the average of ten-10 second Iperf tests. In each figure, a Standard TCP stream and a Scalable TCP stream, each with a different [a,b] pair, were used.

Figure 23. Scalable TCP using a=1/25, b=1/2.

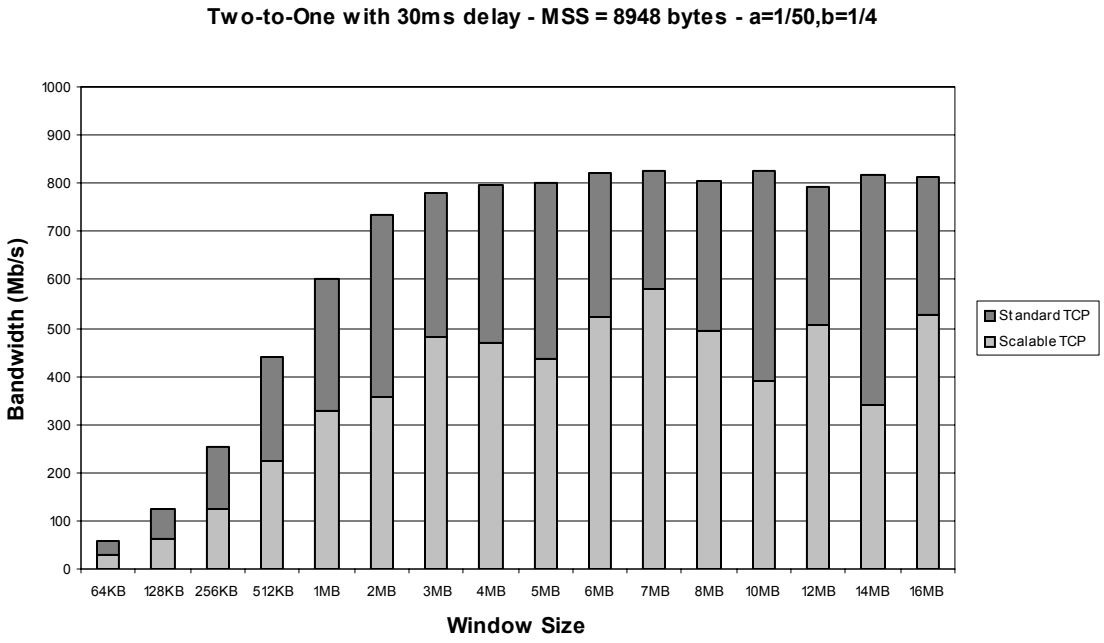

Figure 24. Scalable TCP using a=1/50, b=1/4.

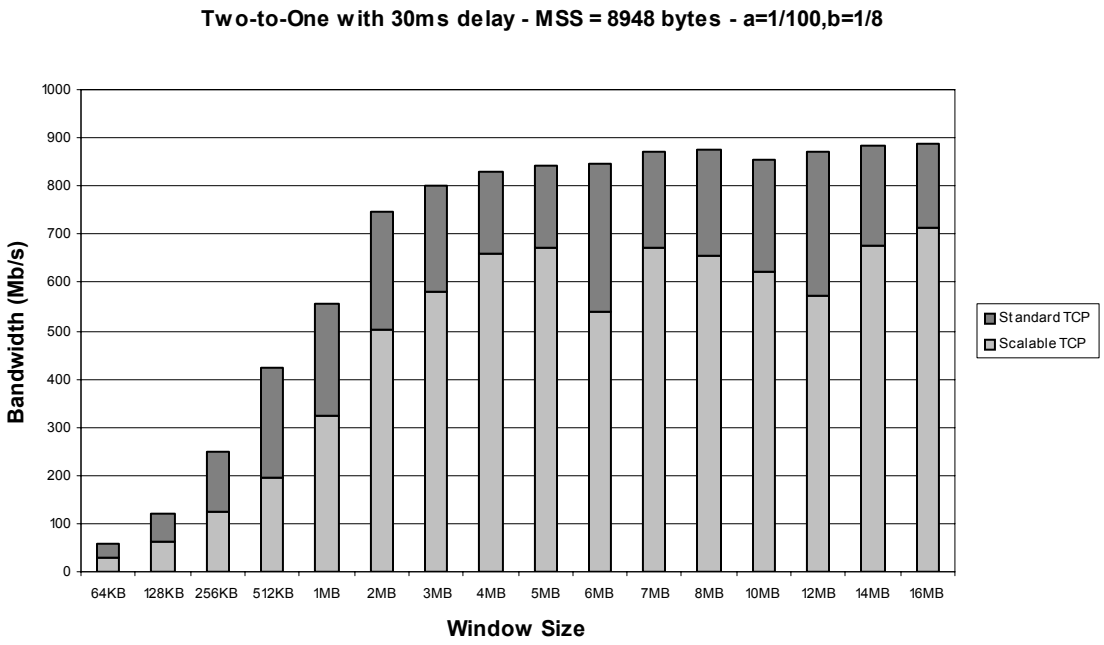

Figure 25. Scalable TCP using a=1/100, b=1/8.

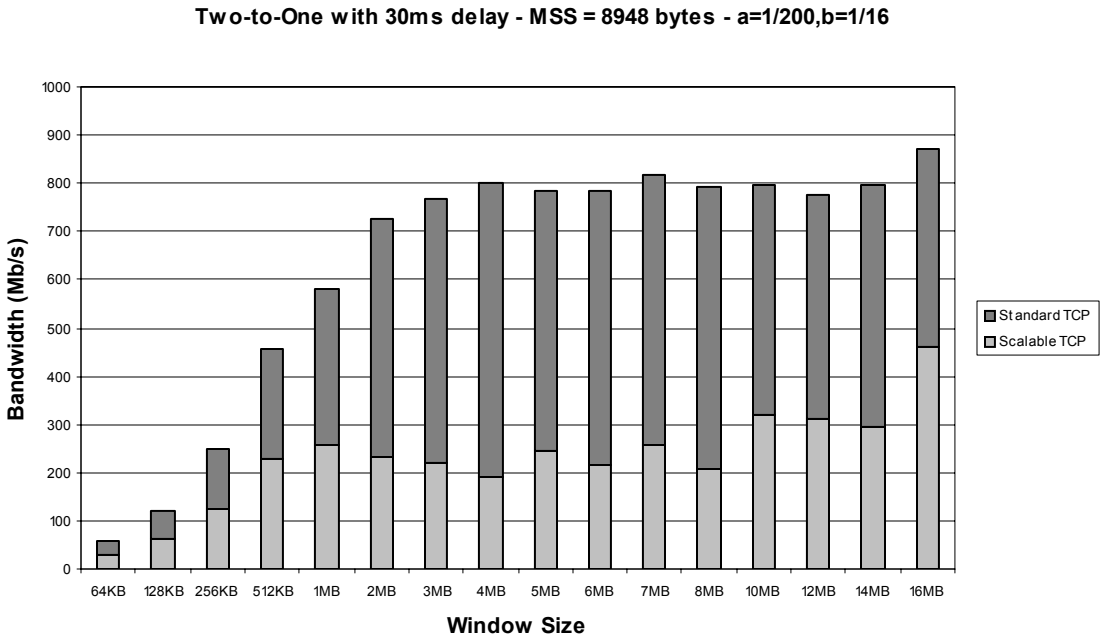

Figure 26. Scalable TCP using a=1/200, b=1/16.

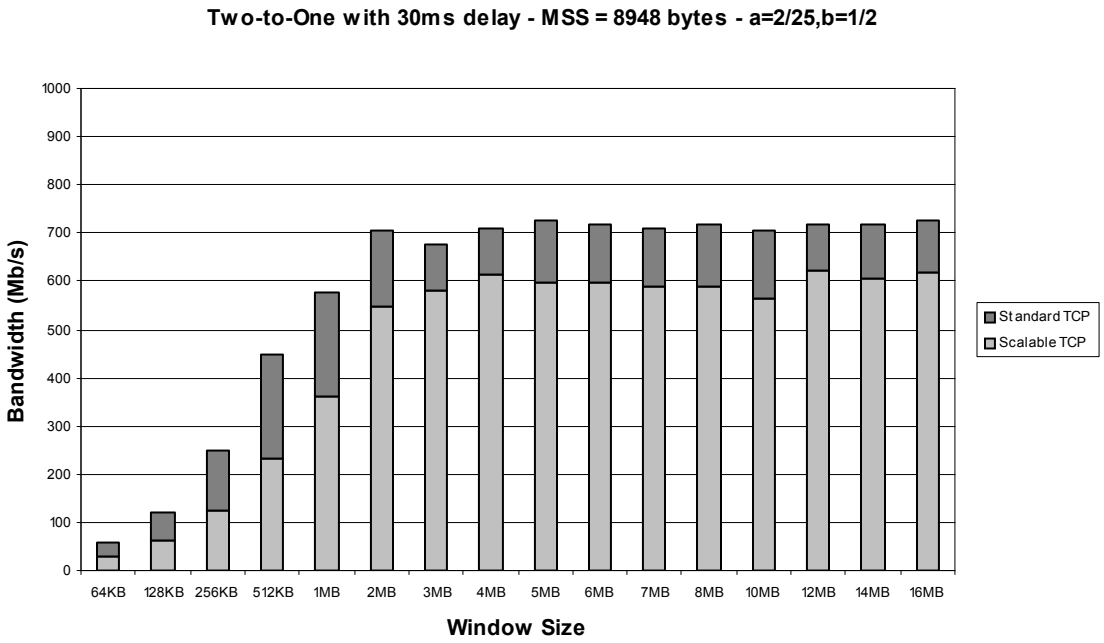

Figure 27. Scalable TCP using a=2/25, b=1/2.

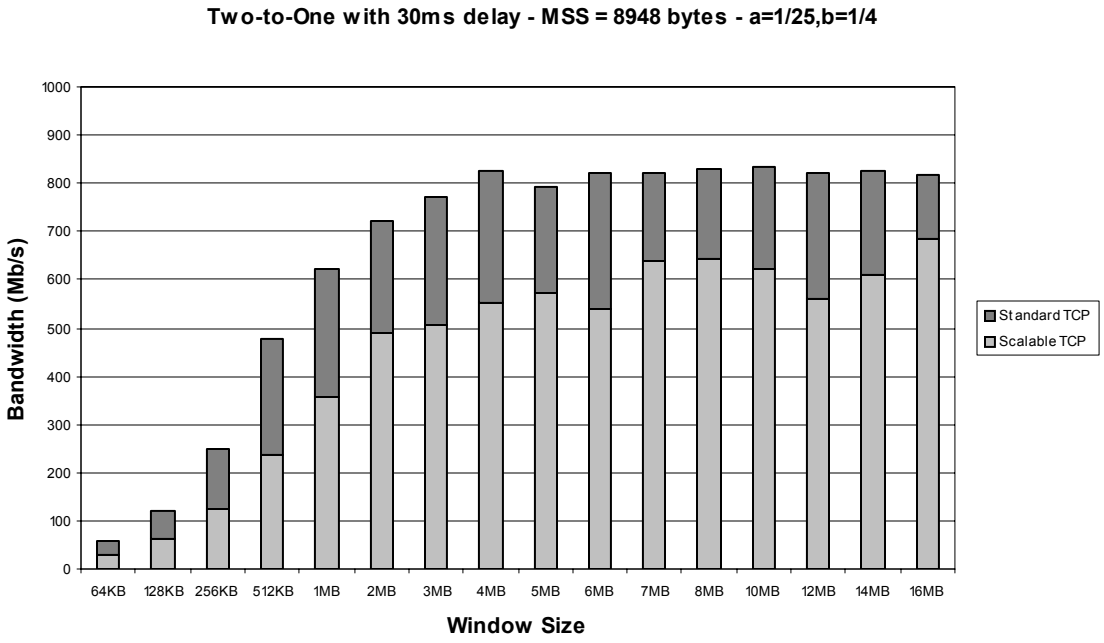

Figure 28. Scalable TCP using a=1/25, b=1/4.

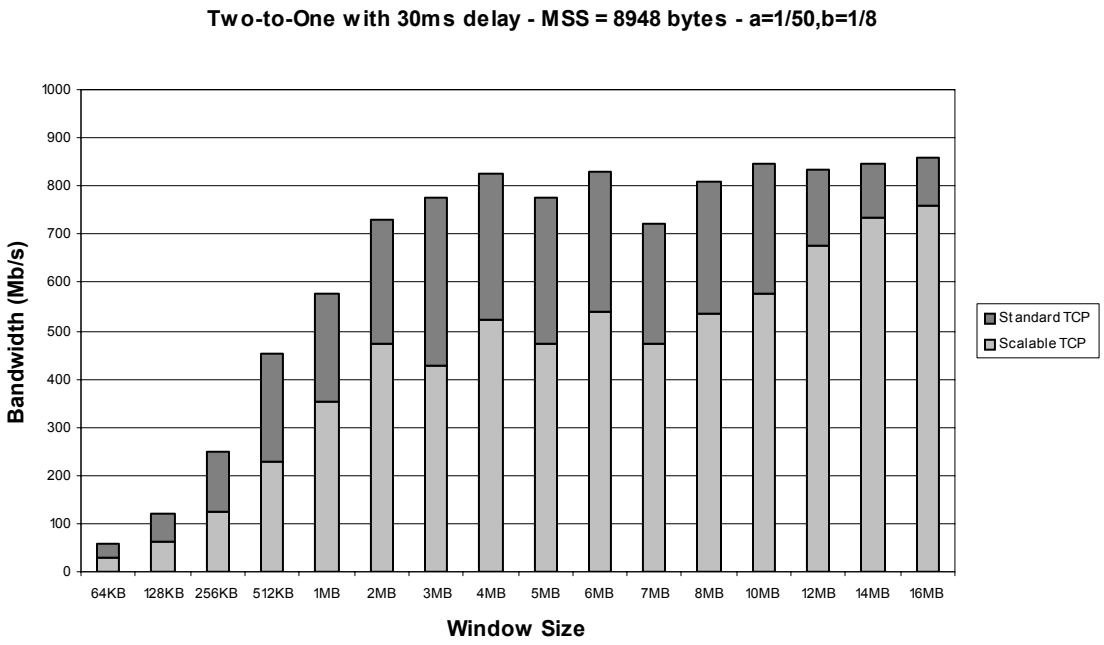

Figure 29. Scalable TCP using a=1/50, b=1/8.

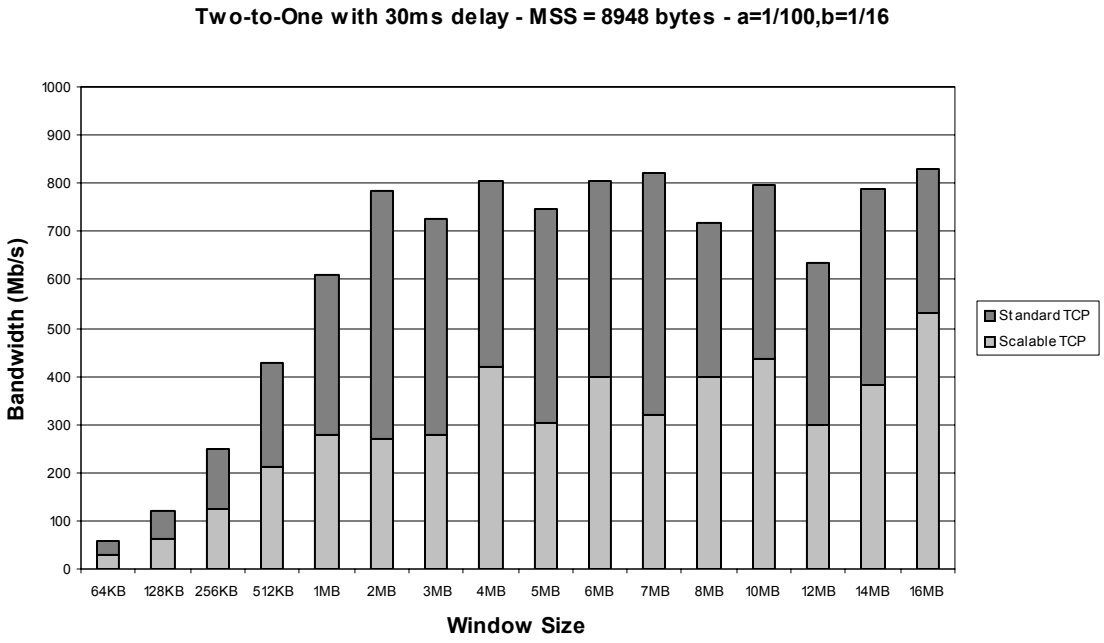

Figure 30. Scalable TCP using a=1/100, b=1/16.

Each [a,b] pair was then ranked based on the results from Figures 23-30. The ranking was based on two variables. The first was the combined throughput of both streams; this had the most impact on the overall rating. The second variable used was Standard TCP's throughput; this was looked at if two [a,b] pairs had similar overall throughput to allow an [a,b] pair that shared more to rank higher. The rankings are shown in Table 3.

Rank	a	b
1	1/100	1/8
2	1/50	1/4
3	1/25	1/4
4	1/50	1/8
5	1/200	1/16
6	1/25	1/2
7	1/100	1/16
8	2/25	1/2

Table 3. Scalable TCP a,b pair rankings.

From Table 3, the [a,b] pair that had been used, [1/50,1/8], was only the fourth best. Switching to [1/100,1/8] would provide higher overall throughput and around the same throughput as [1/50,1/8] for Standard TCP, as shown in Figures 25 and 29.

Txqueuelen Testing

The results of the back-to-back testing showed that the txqueuelen has an impact on the maximum throughput of the test streams. Alternative values of txqueuelen were tested to better observe the impact on each of the TCP congestion control algorithms. The goal of testing various txqueulen's was to find if an alternative value could provide higher throughput without decreasing the current levels of sharing. For Standard TCP, the txqueuelen was varied from 5-250 in increments of five using the back-to-back with delay test setup. The test parameters used were a window size of 16MB and MSS=8948 bytes. Figure 31 shows the throughput results for the various txqueuelen's. The highlighted bar indicates txqueuelen = 100. From Figure 31, using txqueuelen = 100 results in 828 Mb/s while using a value >210 results in 966 Mb/s, an increase of 16%. From the back-to-back with delay test setup, using a txqueuelen > 210 would result in the highest throughput.

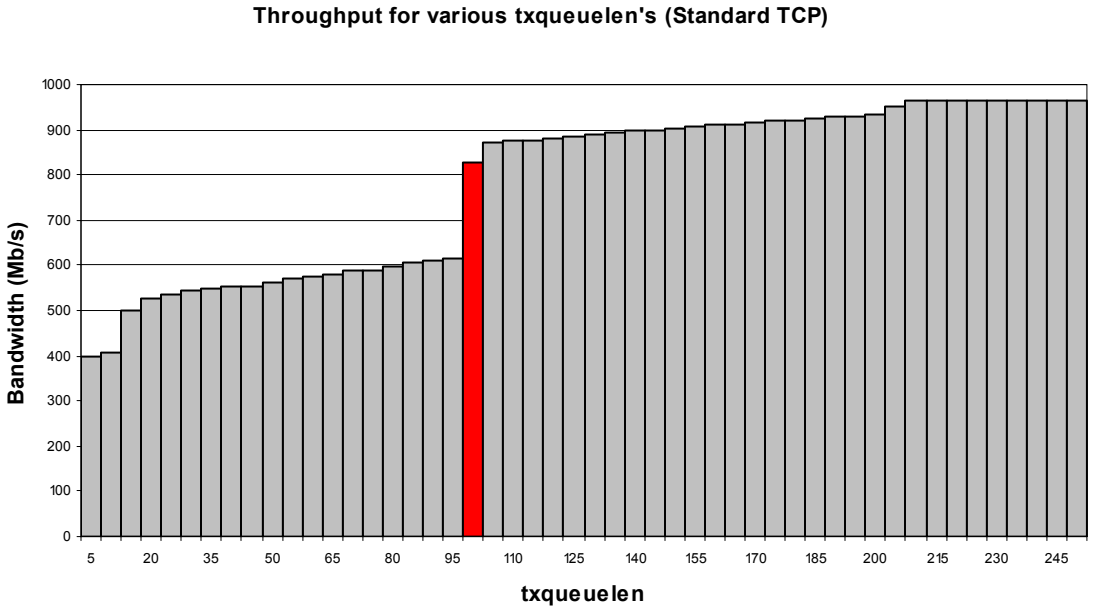

Figure 31. Standard TCP throughput for various txqueuelen's.

To determine what type of effect changing the txqueuelen has on competing streams, Two-to-One tests were performed using different values for txqueuelen. The values used for txqueuelen were 5, 15, 20, 60, 95, 100, 105, 155, 200, 205, 210, and 250. Figure 32 shows the throughput graph using a txqueuelen of 100 and jumbo frames. From Figure 32, eight different window sizes result in a combined throughput of 800+ Mb/s. For txqueuelen = 100, the average combined throughput for window sizes 2MB-16MB is 808Mb/s. Figure 33 shows the throughput graph using a txqueuelen of 250 and jumbo frames. In Figure 33, there are no window sizes that result in a combined throughput of 800Mb/s. For txqueuelen = 250, the average combined throughput for window sizes 2MB-16MB is 767Mb/s, over 50Mb/s, or 5.3%, lower than the average for txqueuelen = 100.

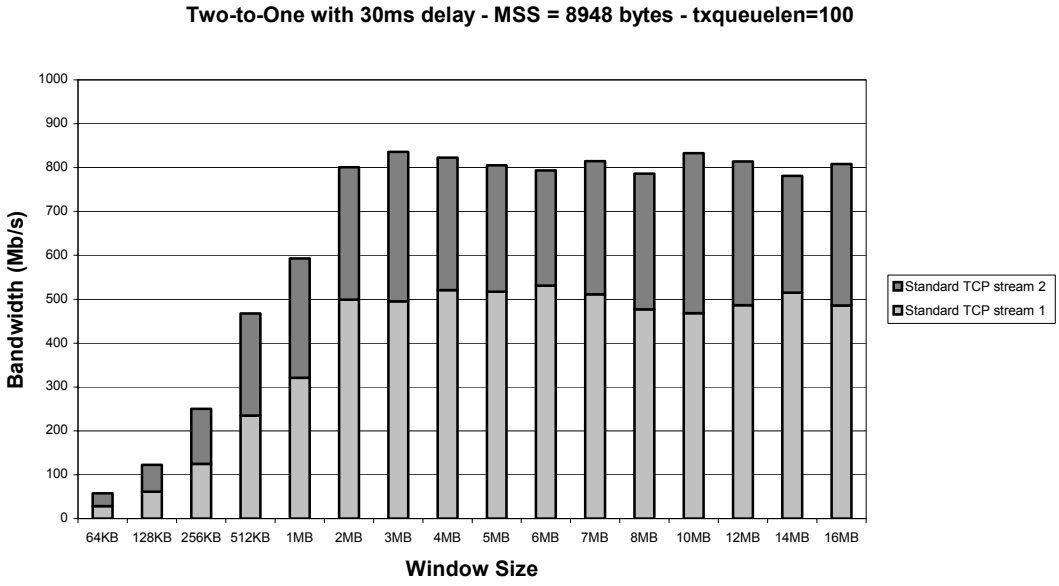

Figure 32. Standard TCP Two-to-One using txqueuelen = 100.

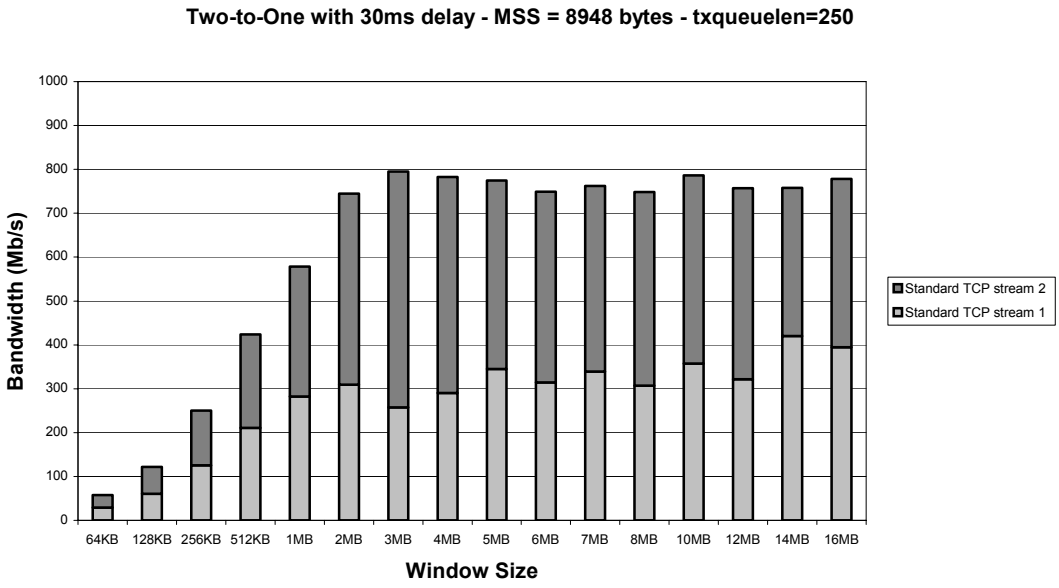

Figure 33. Standard TCP Two-to-One using txqueuelen = 250.

Figure 34 shows the number of window sizes for each txqueuelen tested that result in 800+ Mb/s combined throughput. The maximum number achievable would be 11; combined throughput of both streams resulted in 800+ Mb/s for window sizes in the range of 2MB-16MB (11 different window sizes) for all txqueuelen tested. For Standard TCP, a txqueuelen of 100 provided the highest combined throughput values.

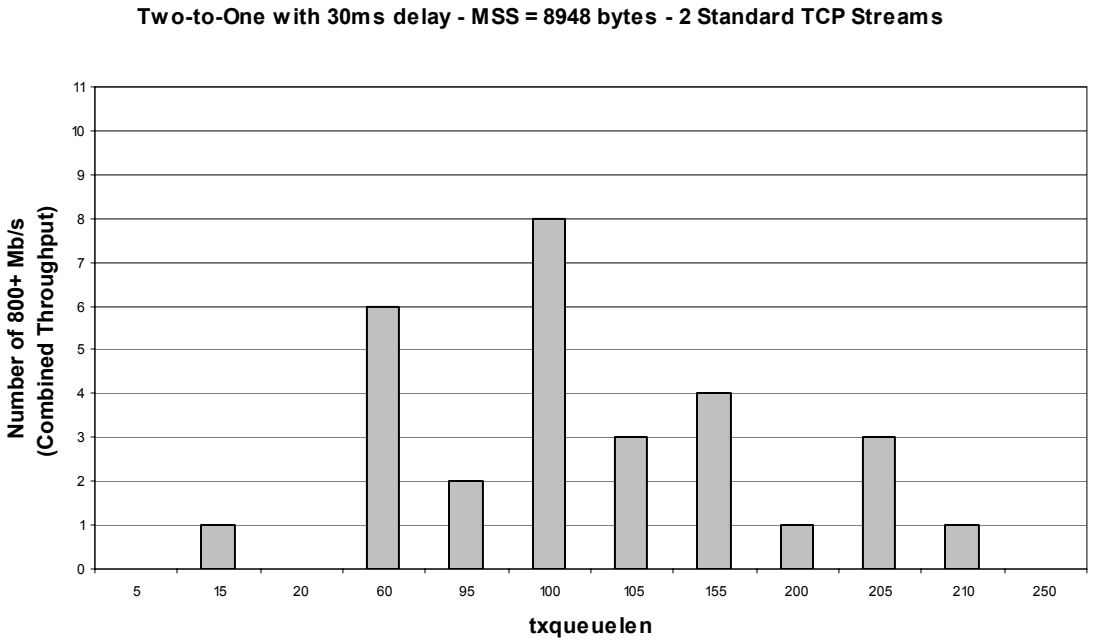

Figure 34. Number of 800+ Mb/s throughputs for various txqueuelen's.

Varying the txqueuelen from 5-250 for Scalable TCP resulted in Figure 35. A 7.7% increase, 897 Mb/s vs. 966 Mb/s, in throughput is gained by using a txqueuelen >210 instead of the default value of 100.

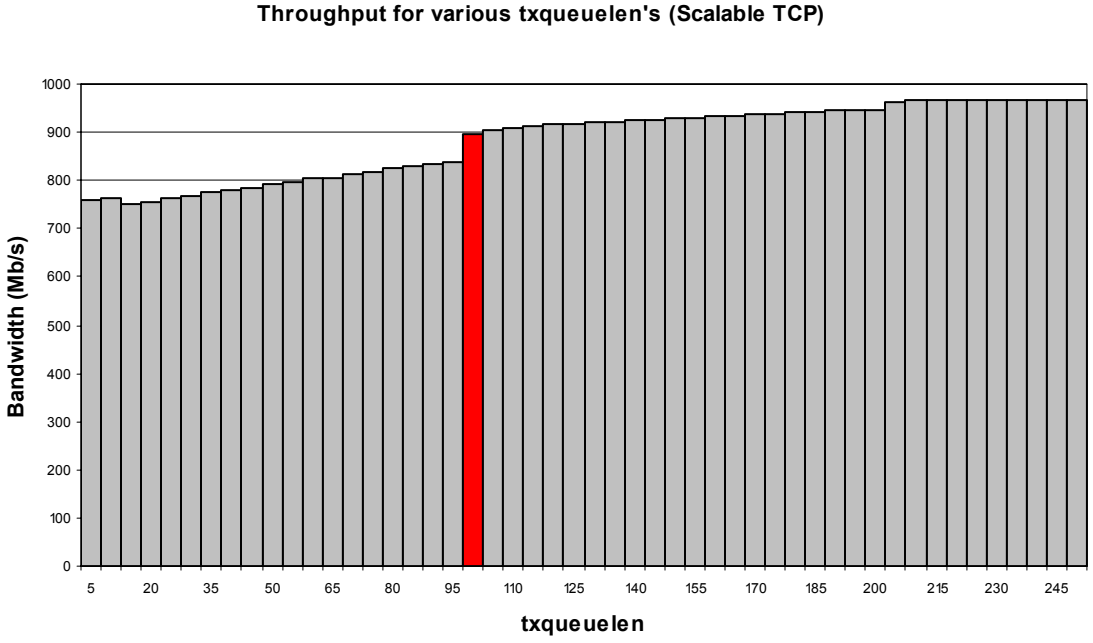

Figure 35. Scalable TCP throughput for various txqueuelen's.

Figure 36 shows the results of Two-to-one testing using an MSS = 8948 bytes and a txqueuelen = 100.

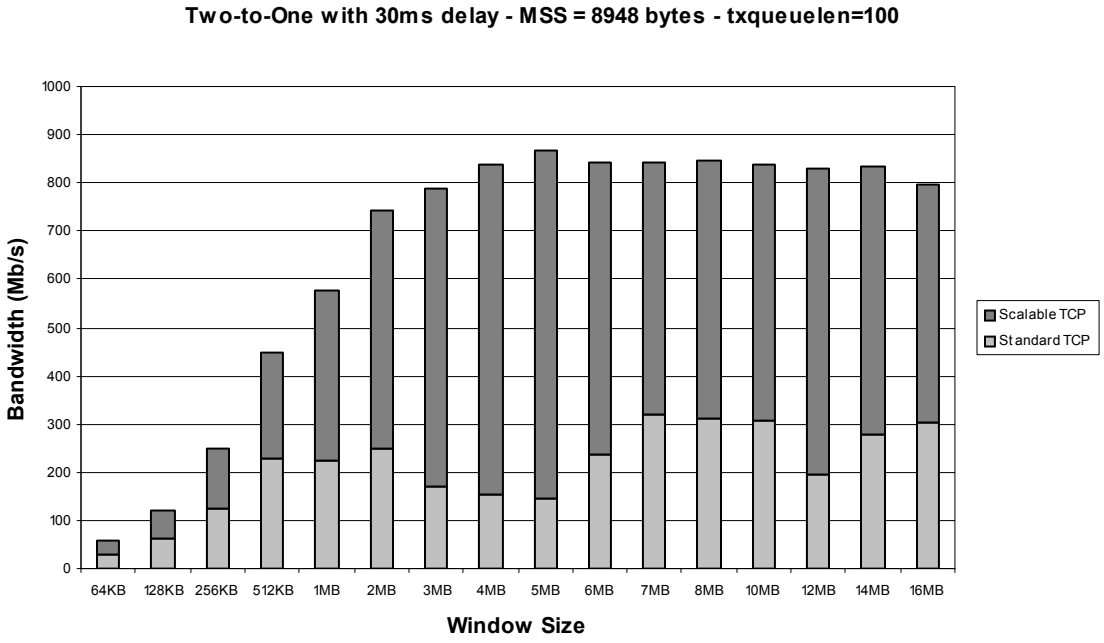

Figure 36. Scalable TCP and Standard TCP Two-to-One using txqueuelen = 100.

Figure 37 shows the results of Two-to-One testing using a txqueuelen of 250.

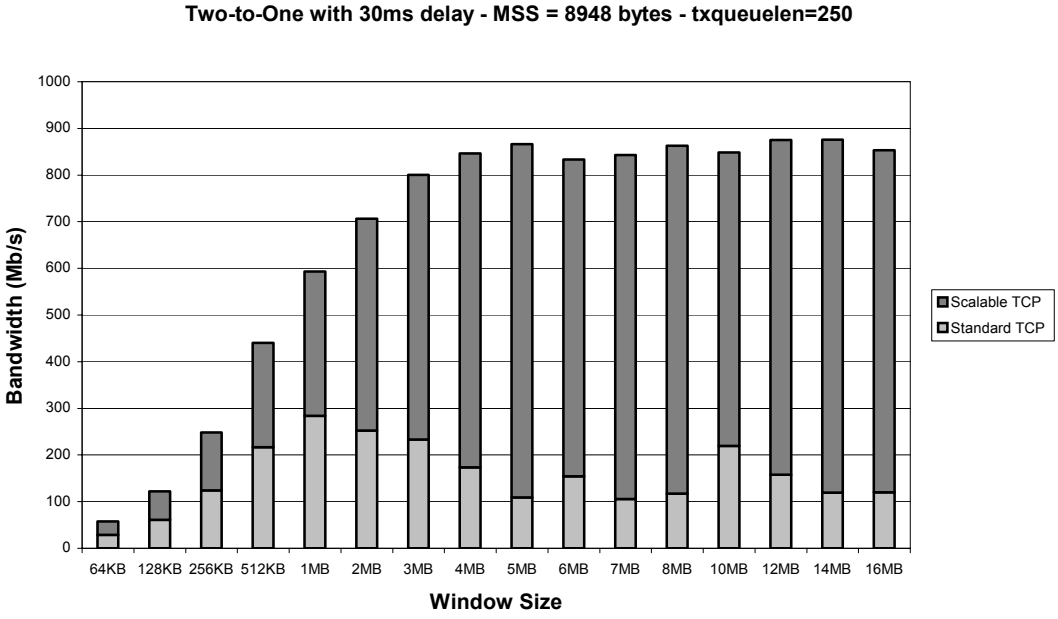

Figure 37. Scalable TCP and Standard TCP Two-to-One using txqueuelen = 250.

From Figure 36 and 37, the increase of txqueuelen resulted in higher overall throughput but lowered the throughput for Standard TCP by almost half.

Figures 38 and 39 show the results of two-to-one testing with two Scalable TCP streams using a txqueuelen of 100 and 250, respectively. The results are similar to those encountered by Standard TCP. The higher txqueuelen of 250 provided lower combined throughput than the default value of 100. The average combined throughput for 2MB-16MB window sizes was 897Mb/s for a txqueuelen of 100. For a txqueuelen = 250, the average combined throughput was 886Mb/s, 1.2% lower than with the default txqueuelen.

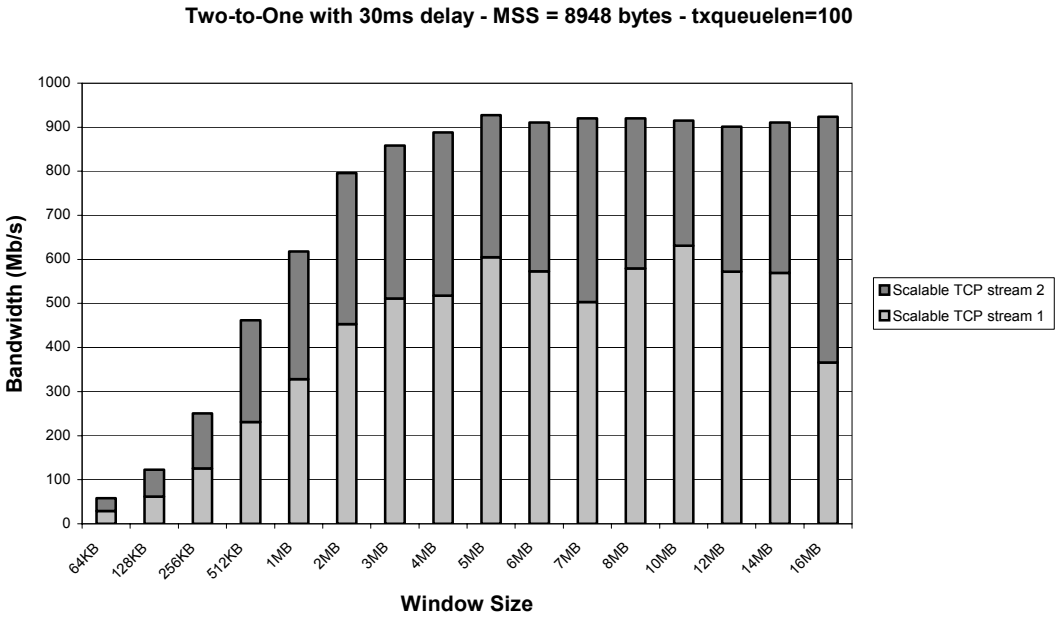

Figure 38. Scalable TCP Two-to-One using txqueuelen = 100.

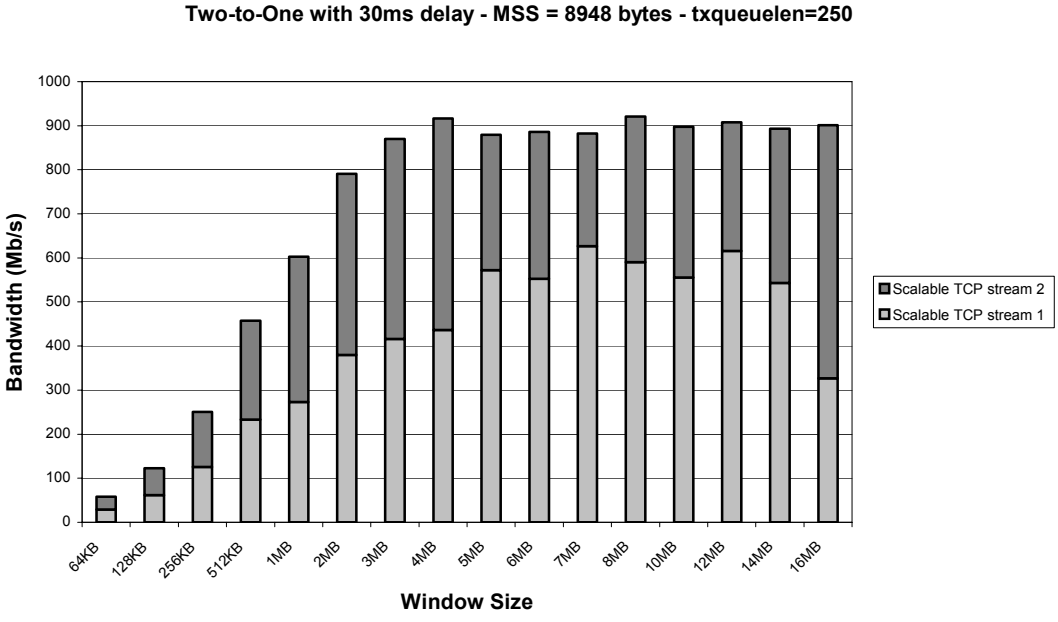

Figure 39. Scalable TCP Two-to-One using txqueuelen = 250.

Figure 40 shows the results of varying the txqueuelen from 5-250 for HighSpeed TCP. A 2% increase, 948 Mb/s vs. 966 Mb/s, is gained by using a txqueuelen >210, over the default value of 100.

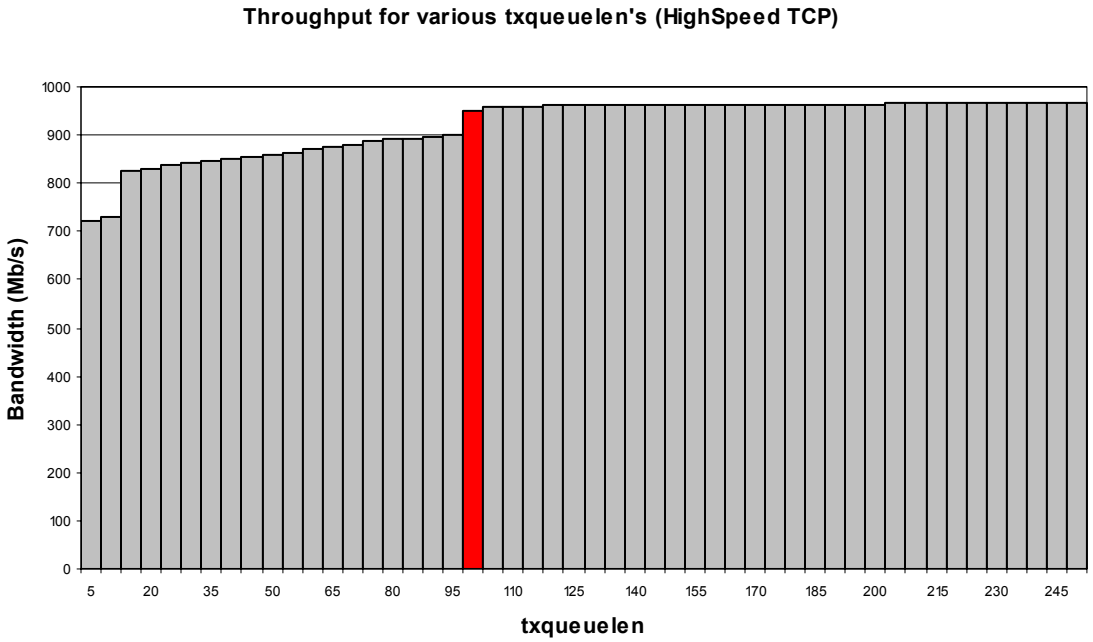

Figure 40. HighSpeed TCP throughput for various txqueuelen's.

The two-to-one testing using HighSpeed TCP and different txqueuelen's showed the same type of effects as Scalable TCP. A txqueuelen of 250 resulted in higher overall throughput while reducing Standard TCP's throughput. These results are shown in Figures 41 and 42.

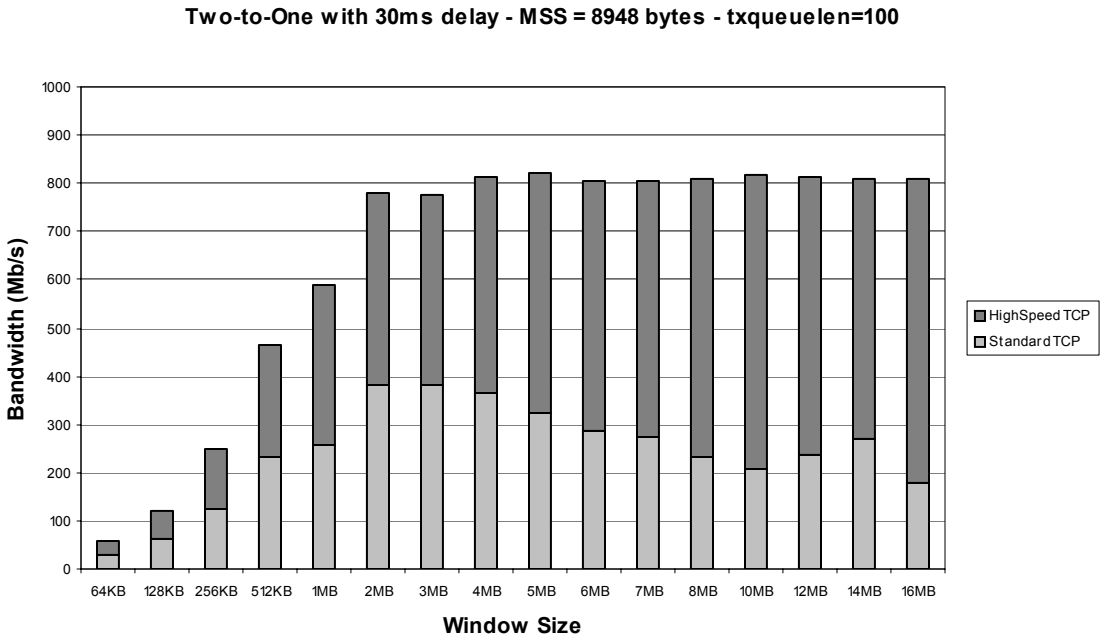

Figure 41. HighSpeed TCP and Standard TCP Two-to-One using txqueuelen = 100.

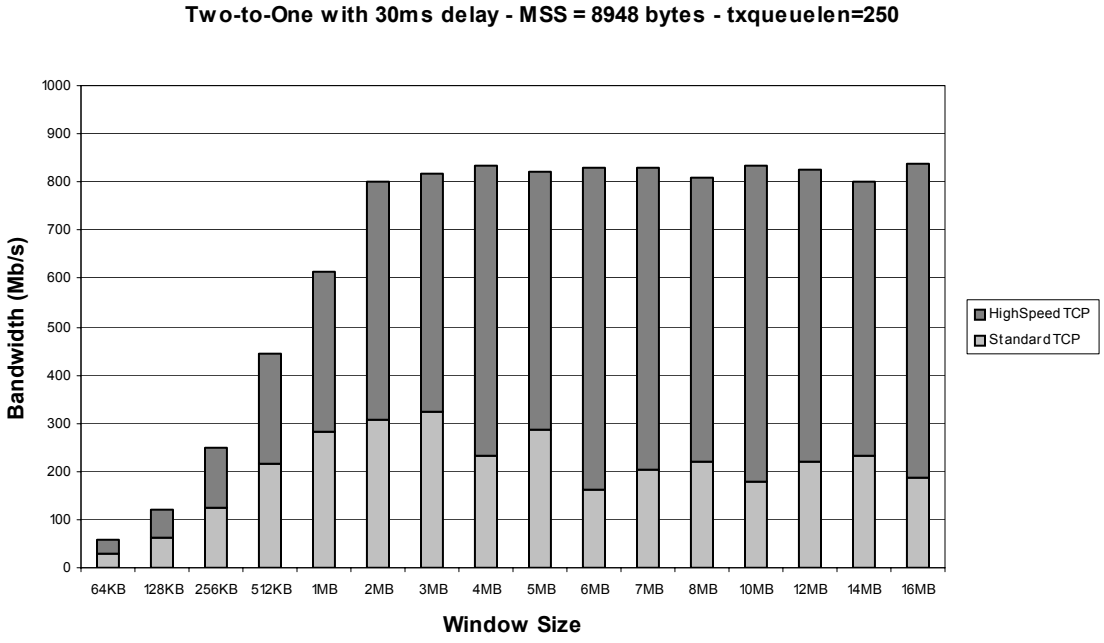

Figure 42. HighSpeed TCP and Standard TCP Two-to-One using txqueuelen = 250.

Figures 43 and 44 show two-to-one tests with two HighSpeed TCP streams and txqueuelen's of 100 and 250, respectively.

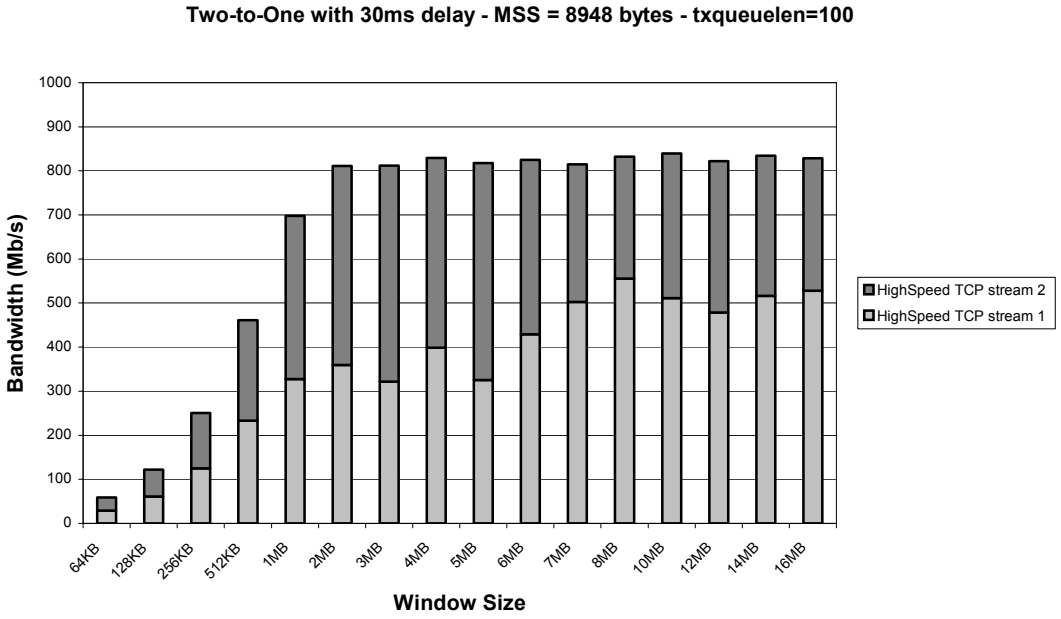

Figure 43. HighSpeed TCP Two-to-One using txqueuelen = 100.

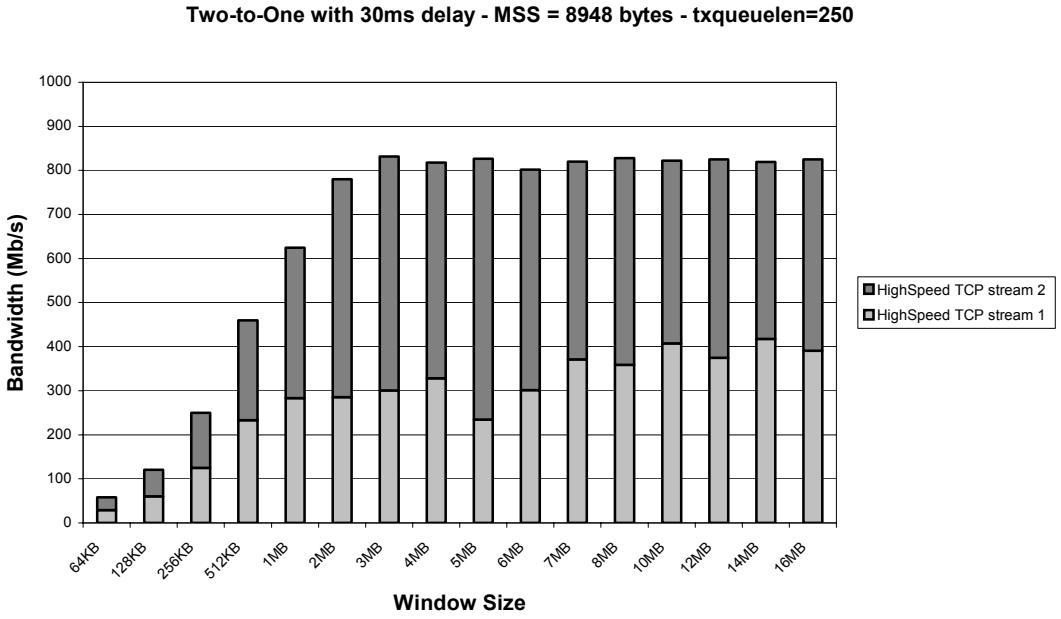

Figure 44. HighSpeed TCP Two-to-One using txqueuelen = 250.

Figures 43 and 44 show that the larger txqueuelen value of 250 again resulted in lower combined throughput values. For HighSpeed TCP, the average combined throughput for window sizes 2-16MB was 824Mb/s for txqueuelen = 100. Using txqueuelen = 250, the average combined throughput was slightly lower at 817Mb/s.

Adjusting the txqueuelen provided higher throughput during the back-to-back testing. However, adjusting the txqueuelen led to lower levels of sharing and decreased throughput in a homogenous environment during two-to-one testing. Based off these results, the default value of txqueuelen = 100 provided the most acceptable test results.

Parallel Stream Testing

Parallel streams are often used to increase network efficiency. The way that parallel streams accomplish the increase in efficiency is as follows. For N parallel streams, a single lost packet only affects a single stream, resulting in a reduction of that streams congestion window. For each of the other N-1 streams, no change is made to the congestion window, so the current throughput for each N-1 streams is maintained.

Parallel streams were tested using Iperf and the back-to-back with 30ms delay setup, Figure 9; Iperf contains a parallel stream parameter specifying the number of streams to use. The number of streams to test was 2,4,8, and 16. Figure 45 shows the total throughput from two parallel for each TCP algorithm used. For two parallel streams, the client in Figure 9 would initiate two separate test streams with the server. The test streams used a MSS = 8948 bytes.

Figure 45. Throughput for Two Parallel Streams.

Figure 46 shows the percent increase over Standard TCP using two parallel streams for each window size using HighSpeed TCP and Scalable TCP. From Figure 46, HighSpeed TCP and Scalable TCP had an increase in overall throughput of 1-7% for most window sizes.

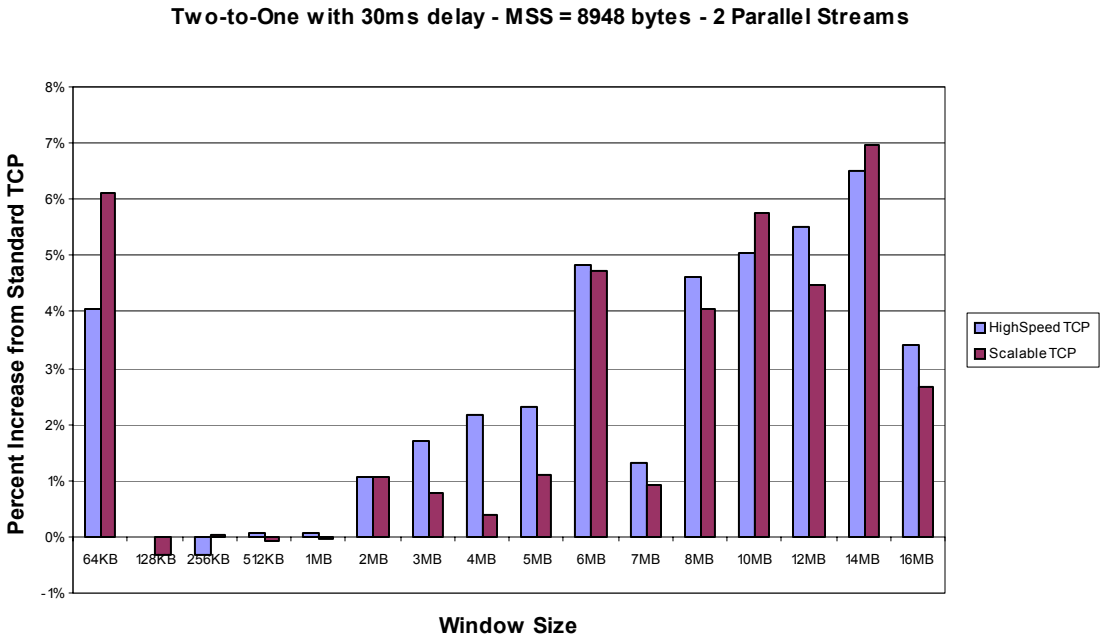

Figure 46. Percent Increase for Two Parallel Streams.

Figure 47 shows the results using sixteen parallel streams.

Figure 47. Throughput for Sixteen Parallel Streams.

Figure 48 shows the percent increase over Standard TCP when using sixteen parallel streams. For sixteen parallel streams, using HighSpeed TCP generally resulted in a decrease in performance of around 0.5-1%. Scalable TCP resulted

in higher decreases than HighSpeed TCP did with some decreases at almost 3% below that of Standard TCP's throughput values.

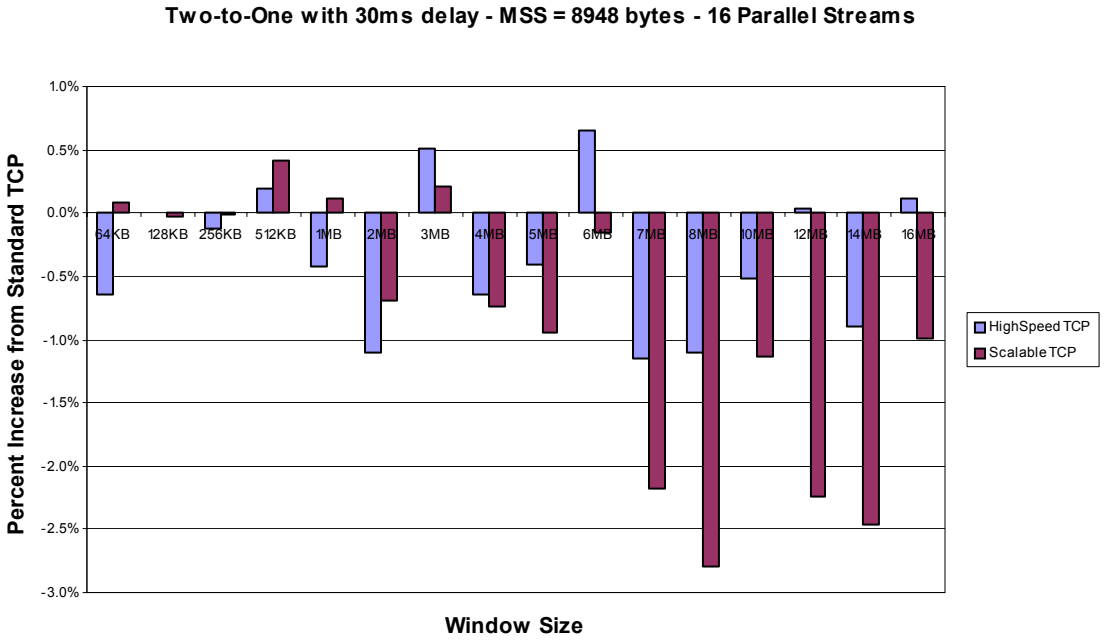

Figure 48. Percent Increase for Sixteen Parallel Streams.

Using parallel streams resulted in little, if any, performance increase by using the alternative congestion control algorithms. For large numbers of parallel streams a decrease in performance was observed when using large window sizes. Parallel streams accomplished their goal and effectively mask the effect of a single drop well enough that Standard TCP performed within 7% of HighSpeed TCP for two parallel streams. Standard TCP even outperforms both HighSpeed and Scalable TCP when using large window sizes for 8, not shown, and 16 parallel streams.

Conclusions

Both HighSpeed TCP and Scalable TCP implement simple changes to the currently used congestion control algorithm. These changes have both a positive and negative effect on existing network traffic. Each alternative algorithm provides higher channel utilization for the high speed, long delay environment. However, the alternative algorithms do not share the channel equally when mixed with Standard TCP traffic. In a homogenous environment, both the overall channel utilization and sharing between streams increases, as compared to a mixed environment. Future work is needed to study the effects of more than two competing streams.

References

[1] W. Noureddine, F. Tobagi. The Transmission Control Protocol, URL http://citeseer.nj.nec.com/noureddine02transmission.html, 2002

[2] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM 1988*.

[3] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control, *Internet RFC 2581*, April 1999.

[4] S. Floyd. HighSpeed TCP for Large Congestion Windows. *Internet Draft* < draft-ietf-tsvwg-highspeed-01.txt>, August 2003. Work in progress.

[5] T. Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks, URL http://www-lce.eng.cam.ac.uk/~ctk21/scalable/, December 2002

[6] High-Speed TCP, URL http://www.hep.man.ac.uk/u/garethf/hstcp

[7] Scalable TCP, URL http://www-lce.eng.cam.ac.uk/~ctk21/scalable/

[8] MAGNET -- Monitoring Apparatus for General kerNel-Event Tracing, URL http://public.lanl.gov/radiant/research/measurement/magnet.html

Distribution

1	MS 0801	M.R. Sjulin, 9330
1	MS 0805	W.D. Swartz, 9329
1	MS 0806	C.R. Jones, 9322
1	MS 0806	Len Stans, 9336
1	MS 0806	J.P. Brenkosh, 9336
1	MS 0806	J.M. Eldridge, 9336
1	MS0806	A.Ganti, 9336
1	MS 0806	S.A. Gossage, 9336
1	MS 0806	T.C. Hu, 9336
1	MS 0806	B.R. Kellogg, 9336
1	MS 0806	L.G. Martinez, 9336
1	MS 0806	M.M. Miller, 9336
1	MS 0806	J.H. Naegle, 9336
1	MS 0806	R.R. Olsberg, 9336
1	MS 0806	L.G. Pierson, 9336
1	MS 0806	T.J. Pratt, 9336
1	MS 0806	J.A. Schutt, 9336
1	MS 0806	T.D. Tarman, 9336
1	MS 0806	J.D. Tang, 9336
1	MS 0806	L.F. Tolendino, 9336
1	MS 0806	D.J. Wiener, 9336
1	MS 0806	E.L. Witzke, 9336
1	MS 0812	M.J. Benson, 9334
1	MS 0826	J. D. Zepper, 9324
1	MS 9018	Central Technical Files, 8945-1
2	MS 0899	Technical Library, 9616 (2)